The Thesis Committee for Michael Stephen Pierce
Certifies that this is the approved version of the following thesis:

Dynamic Thermal Modeling and Simulation Framework: Design of Modeling Techniques and External Integration Tools

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: _____
Thomas M. Kiehne

_____
Carolyn Conner Seepersad

Dynamic Thermal Modeling and Simulation Framework: Design of Modeling Techniques and External Integration Tools

by

Michael Stephen Pierce, B.S.M.E.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2009

"You are not here merely to make a living. You are here in order to enable the world to live more amply, with greater vision, with a finer spirit of hope and achievement. You are here to enrich the world, and you impoverish yourself if you forget the errand."

-Woodrow Wilson

Acknowledgements

The completion of this thesis would not have been possible without the assistance, dedication, support, and guidance of the following people:

My parents, Ben and Marlene, without your love, support, and constant motivation, I would never have become the man I am today. Better teachers, givers, and role models have never existed.

My sister, Sarah, who taught me through example to strive for perfection in everything that I do. I am thankfully each and every day for the guidance, wisdom, and laughter you bring to my life.

My lifelong friends from Waco, Dallas, Austin, and afar, who have always brought joy, humor, and perhaps even a little humility to this lifelong journey. I never would have made it without your constant smiles and encouragement.

My coworkers at the Applied Research Laboratories, Matt Pruske and Pete Backlund, who gave so much more to the completion of this thesis than can ever be repaid.

Countless teachers and professors who have guided me in both life and education, notably Dr. Paul Krueger, Dr. David Willis, Dr. Randall Scalise, Judy Etchison, and Mark Fontenot.

Dr. Seepersad, who graciously agreed to provide insightful contributions throughout my graduate journey. For your patience and dedication I will always be grateful.

And finally, Dr. Kiehne, who believed in me from the day we first met. Without your overwhelming assistance, direction, and constant inspiration, I would never have been able to complete this monumental achievement.

4 December 2009

Abstract

Dynamic Thermal Modeling and Simulation Framework: Design of Modeling Techniques and External Integration Tools

by

Michael Stephen Pierce, M.S.E.

The University of Texas at Austin, 2009

Co-Supervisor: Thomas M. Kiehne

Co-Supervisor: Carolyn Conner Seepersad

In looking to the future of naval warfare, the US Navy has committed itself to development of future classes of an All-Electric Ship (AES) that will incorporate significant technological advancements in the areas of power management, advanced sensor equipment and weaponry, reconfigurability, and survivability systems while simultaneously increasing overall system efficiencies and decreasing the operational costs of the future naval fleet. As part of the consortium responsible for investigating the viability of numerous next-generation technologies, the University of Texas at Austin is dedicated to providing the capabilities and tools to better address thermal management issues aboard the future AES.

Research efforts at the University of Texas in Austin have focused on the development of physics-based, dynamic models of components and subsystems that

simulate notional future AES, system-level, thermal architectures. This research has resulted in the development of an in-house thermal management tool, known as the Dynamic Thermal Modeling and Simulation (DTMS) Framework. The work presented herein has sought to increase the modeling capabilities of the DTMS Framework and provide valuable tools to aid both developers and users of this simulation environment.

Using numerical approximations of complex physical behaviors, the scope of the DTMS Framework has been expanded beyond elements of thermal-fluid behaviors to capture the dynamic, transient nature of far broader, more complex architectures containing interconnected thermal-mechanical-electrical components.

Sophisticated interfacial systems have also been developed that allow integration of the DTMS Framework with external software products that improve and enhance the user experience. Developmental tools addressing customizable presentation of simulation data, debugging systems that aid in introduction of new features into the existing framework, and error-reporting mechanisms to ease the process of utilizing the power of the simulation environment have been added to improve the applicability and accessibility of the DTMS Framework.

Finally, initial efforts in collaboration with Mississippi State University are presented that provide a graphical user interface for the DTMS Framework and thus provide far more insight into the complex interactions of numerous shipboard systems than would ever be possible using raw numerical data.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

In the face of changing global circumstances, the US Navy must be prepared to lead the United States into the future with significant technological advancements in the areas of shipboard power management, advanced sensor equipment and high-powered weaponry, dynamic system reconfigurability, and intelligent shipboard management systems, all while simultaneously decreasing the operational costs and environmental impacts of the future naval fleet. With the continual rise in the cost of fossil fuels and with the instability of the international oil markets, efficiency increases across all shipboard systems have become a driving motivation behind the current design work pursued by the Navy for the next generation of naval ships and submarines. Notably, the Navy has shifted its focus to the design of a broad class of All-Electric Ship (AES), which utilizes electricity as the primary means of energy transport onboard its surface vessels [33].

As background for the work reported in this thesis, the rationale for the Navy's shift to an all-electric fleet is established in this chapter, along with the formation of the Electronic Ship Research and Development Consortium (ESRSC). Additionally, the thermal consequences of electricity-based systems are introduced, with emphasis on the role of the University of Texas at Austin (UT) in this area. Finally, a brief synopsis of the work reported in this thesis is provided.

1.1 THE ALL-ELECTRIC SHIP

Efficiency, reliability, and reduced operational costs are the major motivations behind the Navy's focus on design and implementation of an All-Electric ship architecture for the next-generation naval fleet. In legacy ship designs, the propulsion and electrical power systems operate separately and independently, leading to gross

inefficiencies in terms of both operating capability and spatial layout. For example, over 90% of the total, onboard power capacity of the DDG-51 guided missile destroyer is dedicated solely to the propulsion system. While rated at a maximum speed of 30 knots, the ship rarely exceeds a cruising speed of 20 knots, leading to a staggering 63% of the available power capacity remaining unused under the most common operating conditions [22]. Unfavorable part-load operation of the gas turbine engines used to power both the propulsion and the electrical systems further leads to greater fuel consumption and additional system inefficiencies [14].

The All-Electric Ship seeks to reduce the inefficiencies of legacy ship designs by employing an Integrated Power System (IPS) that combines all aspects of the ship's power systems into a single central grid capable of accommodating propulsion, electrical power, pulsed electrical loads, and energy storage capabilities. This type of power system provides numerous benefits, including [22]:

- Reconfigurability and system redundancies that allow greater ship survivability if aspects of the power system become compromised during combat missions,
- Decreased maintenance requirements due to commonality of ship systems,
- Situational power distribution capabilities that allow numerous systems to function at their optimum operating point, resulting in increased efficiency, and
- Overall system adaptability to enable future high-energy sensor systems and advanced weaponry.

Under the IPS design approach, all prime movers (gas turbines, diesel engines, fuel cells, nuclear reactors, or hybrid systems) generate electrical power that is supplied to the various ship systems via "smart grid" technology. Using technologically adavanced Power Distribution Modules (PDM) and Power Conversion Modules (PCM),

this electrical power is then intelligently distributed to ship propulsion systems, high-powered loads, and continuous service loads throughout the ship. Supplementary energy sources, such as fuel cells, heat recovery units, and brake power recovery systems may additionally feed directly into the electrical grid, while energy storage units, such as flywheels and electric batteries, provide the means to collect any overproduction of electricity for future utilization. The level of intelligent control over the electrical distribution provided by the IPS design has the potential to dramatically increase efficiency and flexibility of the future All-Electric Ship [23].

1.2 ELECTRIC SHIP RESEARCH AND DEVELOPMENT CONSORTIUM (ESRDC)

In 2002, the Office of Naval Research (ONR) formed and funded the Electric Ship Research and Development Consortium (ESRDC) and tasked it with research and development concerning the tools and technologies required to make the All-Electric Ship a reality. Originally conceived with four universities, the consortium has since expanded to include the University of Texas at Austin, Florida State University, Massachusetts Institute of Technology, Purdue University, Mississippi State University, the University of South Carolina, the United States Naval Academy, and the Naval Post Graduate School. Over the past seven years, the primary function of the collaborative efforts of the ESRDC has been to stimulate a multidisciplinary approach to the electric naval force by addressing aspects of system complexity and by developing the necessary modeling and simulation tools for system design and engineering with the goal of reducing the risk and cost of early design decisions [5].

Throughout its research efforts, the ESRDC has recognized the importance of the development and utilization of both commercial and in-house software simulation products in order to provide detailed analysis concerning the electrical, mechanical, and thermal aspects of the design of the All-Electric Ship. Although hardware and software

testing has been employed during the later stages of the design process, cost-effective early-stage decisions are best made using computer-based modeling and simulation tools for the design and optimization of complex ship systems. Recent efforts of the ESRDC have shifted the focus away from commercial software systems toward the development of flexible and accessible in-house simulation tools, as discussed more thoroughly for thermal management in Chapter 3. Sophisticated software tools capable of electrical and thermal dynamic simulation have been developed through the consortium efforts aimed at handling the highly transient conditions dictated by combat as well as the high-energy scenarios mandated by the reconfigurable nature of the electrical grid in the IPS. The development of accessible, dynamic, controls-based software simulating electrical, mechanical, and thermal interactions in a design and optimization sense remains a central mission of the ESRDC.

In order to significantly expand the science and technology base needed by the Navy and industry to successfully build the desired functionality, efficiency, reliability, and cost effectiveness into the AES, the ESRDC is organized into five central development thrusts: computational tools for early ship design, ship electric power system, total ship system solution to thermal management, load management, and next generation integrated power system [6].

1.3 THERMAL MANAGEMENT AT THE UNIVERSITY OF TEXAS AT AUSTIN

With the addition of advanced shipboard electronics, including high-powered radar and sonar sensor suites, high-energy weapon systems such as an electromagnetic rail gun, and other high-power electronic systems, the innovations of the AES will likely produce considerable thermal side effects that could potentially lead to catastrophic failures at the system and component levels [4]. The thermal management requirements of the AES are likely to increase the required shipboard cooling capacity by as much as

700%, thus leading the US Navy to recognize the importance of thermal management as an integral part of the design process and as an enabler for all of the other advanced technologies of the future naval fleet.

While legacy ship designs have employed simplistic and often ad-hoc approaches to the design of the thermal management system, these methods are ineffective for the future that the Navy wishes to pursue. As part of the ESRDC, the research team at the University of Texas at Austin has spearheaded a research effort focused on thorough understanding of thermal management strategies at the shipboard system level. Since the inception of the ESRDC in 2002, seven UT students have completed their graduate degrees under the thermal management thrust of this consortium. Initial work involved the modeling of thermal aspects and impacts of the power-generation and high-powered electronics systems using commercial software products. Mr. Brian Carroll completed work in modeling fuel cell/gas turbine hybrid propulsion and developing a transient fuel cell model [3]; Mr. Scott Haag utilized the commercial power modeling tool *ProTRAX* to construct a dynamic chiller model for the purposes of simulating startup and partial-loading conditions [10]; Mr. Ty Webb investigated the electrical and thermal dynamics of pulsed weapons technology, specifically the electromagnetic rail gun [29]; and Mr. Christopher Holsonback modeled the electrical-mechanical-thermal characteristics of an integrated electric propulsion system using *ProTRAX* [14].

1.4 DYNAMIC THERMAL MODELING AND SIMULATION (DTMS) FRAMEWORK

With the consortium shifting away from the use of commercial simulation tools, Mr. Patrick Paullus completed work in December 2007 [20] on the construction of an in-house network simulation tool, subsequently known as the Dynamic Thermal Modeling and Simulation (DTMS) Framework, capable of simulating the complex thermal loads and shipboard cooling systems required to continue research on the thermal management

aspects of the AES. While laying the groundwork for future simulation efforts at the University of Texas, Mr. Paullus utilized the newly created framework for the simulation of the starboard freshwater chilling loop onboard the DDG-51 destroyer.

Since initial development of the framework, several UT students have continued modeling work under the thermal management thrust of the ESRDC by utilizing the power and customization options provided in the DTMS Framework. In the fall of 2008, Mr. Patrick Hewlett completed a simulation of the York 200-ton chiller used onboard the DDG-51 by developing models of two-phase heat exchangers and dynamic control systems [13]. In the summer of 2009, Mr. Matthew Pruske also completed work utilizing the DTMS Framework for a thermal-electrical co-simulation of the shipboard integrated power systems of a notional AES [23].

1.5 THESIS ORGANIZATION

Development of the DTMS Framework has continued beyond the initial groundwork presented in the thesis of Mr. Paullus to advance the software environment beyond an academic research tool into a highly-developed design and simulation product for use both by the US Navy and its industrial partners. This thesis documents these advances in the simulation architecture of the DTMS Framework, including the addition of complex simulation methodologies to allow the simulation of a broader range of physical systems, the creation of a sophisticated interfacial system, and the improvement of the methods for extracting meaningful data from DTMS simulations, This work culminates with the first example of the integration of the DTMS Framework with an external software program, FireGUI.

While the simulation architecture of the DTMS Framework is based on well-known, widely-used simulation methods, a complete discussion of the modeling methodology utilized by the framework has not previously appeared in the published

literature independent of discussion of the DTMS Framework itself. Chapter 2 provides a discussion of the Resistive Network modeling technique using comparisons to electrical networking and bond graph theory. This chapter also demonstrates the process of preparing physical equations for use in the DTMS Framework and details the advantages of Resistive Network modeling methods over Resistive Companion modeling techniques.

Chapter 3 introduces the C++ principles that form the backbone of the DTMS Framework and describes the layout and C++ class hierarchy of the various base classes and subclasses that form the primary components of the framework. Chapter 4 documents the addition of capacitive and inertial modeling components to the DTMS Framework, including the numerical methods utilized to incorporate these components into the Resistive Network modeling strategy. A simple linear RLC electrical circuit example is constructed in the DTMS Framework using the capacitive and inertial models to demonstrate their performance. This chapter also presents a container model created for the DTMS Framework that allows a single DTMS model to encapsulate the physical behaviors of multiple subcomponents.

In Chapter 5, the design and implementation of a universal input system for the DTMS Framework is presented. As part of this system, a replacement class for the enumeration used to communicate data between various aspects of the simulation system is created to enhance the ease-of-use and independent development principles of the DTMS Framework. Pluggable C++ object classes are also presented to handle the process of dynamically creating DTMS objects during program execution. Finally, generalized input data structures are presented that allow the input system to remain consistent as the DTMS Framework is expanded and improved.

Chapter 6 demonstrates the addition of new methods for extracting meaningful data from a DTMS simulation. Details include discussion of improvements made to the

output system to centralize common system functionality, prevent potential coding errors, and provide additional user customization features. The standardized debugging system added to the DTMS Framework is also discussed in this chapter, along with the exception classes that have been added to improve the developer experience.

Chapter 7 presents preliminary work to demonstrate the integration of the DTMS Framework with the FireGUI graphical interface developed by Mississippi State University. In the process, the contributions of each of the various DTMS technologies presented in the previous chapters of this thesis are discussed as they pertain to the design of this system. DTMS simulation of a heavily-simplified vapor-compression water chiller developed using the FireGUI interface is presented in this chapter.

Finally, chapter 8 concludes the work of this thesis by presenting the path forward for the use of the tools presented in this document and proposed directions for further development of the DTMS Framework.

Chapter 2: Resistive Network Modeling Strategy

In developing the in-house software framework known as the DTMS Framework that forms the basis of this thesis document, a primary modeling methodology was selected in order to provide a foundation for current and future work of the thermal management team at the University of Texas. While the DTMS Framework is capable of incorporating and efficiently managing any number of modeling strategies, it was decided to centralize the team efforts around a common modeling strategy that would provide a platform for consistent model development and provide a complete example for future developers to build upon with additional modeling strategies as desired.

Taking the lessons learned from the commercial software that had been utilized previously by the thermal management team at the University of Texas, the Resistive Network modeling strategy was chosen as the primary modeling system for use in the DTMS Framework due to its applicability to numerous energy domains, component-independent modeling philosophy, multiple layers of detail, and straightforward solution methods. Employing comparisons to electrical networks and bond graph theory, this chapter describes the Resistive Network modeling approach and discusses the benefits of this modeling strategy for system-level simulations. A fluid flow example is constructed to demonstrate how standard equations are utilized in Resistive Network modeling, and a comparison between Resistive Network modeling and Resistive Companion modeling is presented.

2.1 General Modeling Methodology

In engineering, the modeling of a system involves constructing an approximation of a physical system using constituent equations that govern the system's behavior. The level of detail provided by a model is governed primarily by the equations utilized and

the assumptions that are made concerning behavior of different aspects of the system. A modeling strategy involves organizing these constituent equations into a consistent format that can then be solved given a known set of simulation parameters in order to provide meaningful information about the physical system that is being modeled.

The Resistive Network modeling approach requires that the constituent equations of a system be organized such that they represent resistive loads on a network. Most commonly, this is demonstrated by using an analogy with electric circuits. Although Resistive Network modeling itself can be applied to systems of many different energy domains (electrical, thermal, fluid, mechanical, rotational, etc.), the basic electrical circuit diagram is frequently used as a simple demonstration to illustrate the modeling strategy.

A resistive network in the electrical energy domain consists of a series of resistors connected together in a network with one or more voltage or current sources, as demonstrated in Figure 2.1:

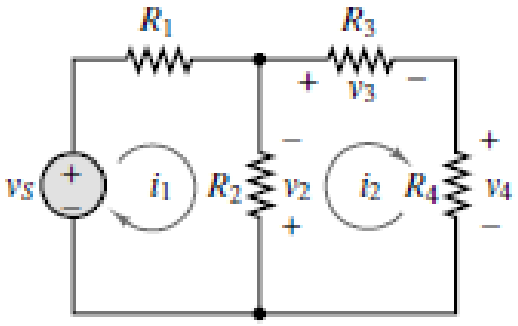


Figure 2.1: Example of an electrical resistive network

The diagram contains three types of components: resistors, junctions, and sources. These are the primary building blocks of the Resistive Network modeling strategy, and each will be described in the following sections.

The remainder of this thesis will utilize the terms “flow” and “effort” to describe different system components in an energy-independent manner. These terms are used

primarily in power-conserving modeling techniques which include Resistive Network modeling. Specifically, in any energy domain, the product of the flow term and the effort term equal the power, e.g., in electrical systems, the flow term is current and the effort term is voltage; in mechanical systems, flow is velocity while effort is force; and in hydraulic systems, flow is volumetric flow rate and effort is pressure.

2.1.1 Flow-Based Components

In a resistive network, flow-based models represent physical components which have a quasi-steady flow rate through the device that is driven by a difference in the efforts between inlets and outlets. At each point in time, the amount of flow entering a model through the inlet must be identical to the amount of flow exiting the model through the outlet. While the value of the flow rate may vary between successive time steps, there can be no storage of flow within a flow-based model. Components that can be represented by this type of model conform to the following general flow-model equation:

$$f = \phi\left(e_{i,1}, e_{i,2}, ..., e_{o,1}, e_{o,2}, ...\right) \tag{2.1}$$

The generic function ϕ is composed of each of the inlet efforts $\left(e_{i,1}, e_{i,2}, ...\right)$ and the outlet efforts $\left(e_{o,1}, e_{o,2}, ...\right)$ associated with the model, and these produce the component flow for the entire model, *f*.

Returning to the electrical circuit example, the constituent equation for a linear electrical resistor is shown below:

$$i = \frac{\Delta V}{R} \tag{2.2}$$

where i is the current through the resistor, ΔV is the voltage drop across the resistor, and R is the linear resistance. Comparing this with the general flow-model equation (2.1), i

represents the flow f, ΔV represents the difference between the inlet effort $e_{i,1}$ and outlet effort $e_{o,1}$, and $\frac{\Delta V}{R}$ represents the generic function ϕ. Based on this linear resistance equation, the electrical resistor can conform to the general flow-model equation, and therefore it can be considered a flow-based equation.

2.1.2 Effort-Based Components

Effort-based models in a resistive network represent components which have a single effort throughout the model and maintain conservation of flow across the inlets and outlets. Primarily, these models are governed by the following general effort-model equation, based on flow conservation:

$$\sum_{j=1}^{m} f_{i,j} - \sum_{k=1}^{n} f_{o,k} = 0 \tag{2.3}$$

where $f_{i,j}$ is the j^{th} inlet flow and $f_{o,k}$ is the k^{th} outlet flow. Similar to the flow-based components of the Resistive Network modeling strategy, this equation requires quasi-steady flow throughout the effort-based model, which in turn requires that quasi-steady flow be maintained throughout the entire system. In many systems, particularly thermal-fluid systems, this limitation is inconsequential to achieving the required simulation result; however, it can become overly limiting in other systems, particularly electrical systems. Chapter 4 addresses this limitation by creating time-dependent approximations for inertial and capacitive effects that allow a wider range of systems to be modeled using the Resistive Network modeling strategy.

As shown in the electrical example of the previous section, flow-based models require their inlet and outlet efforts to be specified by the bounding models in order for the flow to be calculated. Because of this, two flow models cannot be directly connected

together since neither would be able to provide the effort for the other. The effort-based model provides the necessary data required to complete the model connections of a simulated system. Most often, effort-based models are used solely for this purpose and do not represent any physical component, but exist merely as a modeling construct necessary for solving the system. However, while commonly used for connecting flow models, effort-based models can also represent certain physical models such as mixing chambers, combustion chambers, or other flow junctions. Regardless the vast majority of physical components can be represented with flow-based models.

In the electrical circuit diagram shown in Figure 2.1, the node connecting resistor R3 to resistor R4 and the two wire junctions on either side of resistor R2 are represented by effort-based models in the Resistive Network modeling strategy. As an example, the effort equation for the flow junction the separates resistor R1 from resistors R2 and R3 would be calculated as follows:

$$i_1 - (i_2 + i_3) = 0 \tag{2.4}$$

where i_1 represents the current flowing through resistor R1, i_2 represents the current flowing through resistor R2, and i_3 represents the current flowing through resistor R3.

2.1.3 Source Components

The final modeling component found in Resistive Network modeling is the source component, which can provide either a flow or effort that varies independently of the components to which it is connected. Flow sources are similar to flow-based models except that their flow cannot be a function of the inlet and outlet efforts. A flow source also need not be connected to both an inlet and outlet effort model, as it only requires a single connection to the system. Similarly, an effort source provides an effort to the

system which is calculated independently of the flows entering or exiting it. This feature allows conservation of flow to be violated at effort sources, thus allowing the behavior of external system conditions to be simulated at the boundaries of the system without requiring detailed models of the external systems themselves.

The left side of the electrical circuit diagram in Figure 2.1 contains a voltage source which drives current flow through the remainder of the circuit. Since voltage represents an effort in the electrical energy domain, this voltage source is represented by an effort source in Resistive Network modeling. In a physical system, this voltage source could represent far more complex components such as generators, batteries, or flywheels. But for this simple system, the only important consideration is the source behavior of providing a constant voltage to the circuit. Source elements in Resistive Network modeling allow the behavior of complex external components to be reproduced without the need for reproducing the complete physics of the external components.

2.2 Relating Resistive Network Modeling to Bond Graph Theory

While the electrical circuit analogy is beneficial for introducing the elements of the Resistive Network modeling strategy, it lacks applicability to general modeling techniques due to its close association with electrical networks. It is desirable to utilize a more general modeling approach to describe physical systems, which is difficult to accomplish using strictly electrical circuits.

Bond graph theory is a modeling technique which shares many characteristics with Resistive Network modeling, and is much better suited for graphically modeling a physical system for ultimate implementation in the DTMS Framework. Both bond graph theory and Resistive Network modeling are based on the principles of power transmission and energy conservation, making it extremely easy to convert models from one strategy to the other. Each strategy is designed to approach systems from an energy domain-

independent standpoint and to easily allow for the transfer of energy from one energy domain to another. Each modeling strategy also allows modeled components to maintain their own set of physics-based equations, promoting independent model development, while allowing general system solvers to handle connections between the models in a generic and universal sense. Because of these attributes and due to the close relationship with Resistive Network modeling, bond graph theory has become the primary modeling tool used within the thermal management team at the University of Texas at Austin when seeking to simulate complex physical systems. Many supporting documents related to the DTMS Framework make heavy use of bond graph theory in their component modeling and in their description of the framework.

While bond graph theory is an extremely useful tool for model descriptions and theory discussions, it does have several downsides which led to the ultimate selection of Resistive Network modeling as a more promising alternative for the DTMS Framework. In particular, bond graph theory is more complicated than Resistive Network modeling with respect to the various types of models that are available to the designer, the various connections that can be made between models, and the solution techniques that are required to accurately resolve a system solution. The following sections describe bond graph theory in more detail and present the bond graph equivalents of several elements of Resistive Network modeling.

2.2.1 Bond Graph Methodology

Like Resistive Network modeling, bond graph theory is based on the principle of power transmission between components and the concepts of “flow” and “effort”. Each bond graph component is provided either a flow or an effort from its bounding models and uses its constituent equations to provide the connected models with the complimentary property.

There are six basic model types available in bond graph theory: resistive elements, inertial elements, capacitive elements, source elements, transformers, and gyrators, each of which has specific equations that define its behavior. In general, these components alone do not increase the complexity of bond graph theory over that of Resistive Network modeling. In fact, each of the model types in bond graph theory can be represented in Resistive Network modeling either through time-based approximation of Resistive Network elements or by using combinations of Resistive Network elements. For additional details concerning the simulation of bond graph elements in Resistive Network modeling, refer to Chapter 4 of this thesis and the thesis of Matthew Pruske [23].

The true complexity of bond graph theory, and the reason it was not selected as the primary model strategy for the DTMS Framework, comes from the connections that are allowed between models. Every connection in bond graph theory has both a direction and a "causality" that determines which model is responsible for providing the flow and which is responsible for providing the effort to the other model. This causality is not inherent to the type of element itself and will vary between systems, therefore requiring multiple sets of constituent equations based on the construction of the particular system being modeled. Bond graph theory also relies on two different types of junctions to allow for proper power transfer between the various components: "1-junctions" provide a constant flow to each connected branch while enforcing conservation of effort, and "0-junctions" provide a constant effort to each connected branch while enforcing conservation of flow across each inlet and outlet branch.

These numerous connection possibilities, along with the myriad of special case conditions that arise, make bond graph theory a poor selection for the foundation of a modeling and simulation strategy in the DTMS Framework. However, as stated before,

the bond graph approach has immense usefulness in discussion of theory and description of models, and thus it is appropriate to demonstrate how the three elements of Resistive Network modeling can be represented in bond graph theory using simplified constructs, as is demonstrated in the following sections.

2.2.2 Flow-Based Components

Representing flow-based components of the Resistive Network modeling strategy in bond graph theory is relatively simple due to the direct relationship between this component and the resistive element in bond graph theory. The resistive element, portrayed as the letter 'R' shown in Figure 2.2, represents a model that has an algebraic relationship between its flows and efforts.

Figure 2.2: Bond graph of resistive element

In order to make connections in the DTMS Framework more consistent, a flow-based element is defined as having only a single inlet effort and a single outlet effort, as shown in Equation 2.5:

$$f = \phi_R(e_i, e_o) \tag{2.5}$$

To enforce this single-inlet/single-outlet configuration, the bond graph equivalent of the flow-based component must include the connections that can be made with it. The original design of the flow-based component in the DTMS Framework defined the flow as a function of the difference between the inlet and outlet efforts:

$$f = \phi_R(e_i - e_o) = \phi_R(\Delta e) \tag{2.6}$$

Figure 2.3 below demonstrates how the original DTMS flow-based component would be constructed using bond graph elements:

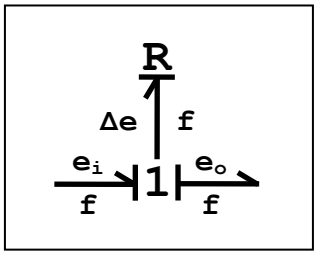


Figure 2.3: Bond graph equivalent of original DTMS flow-based component

The causalities of bond graph connections require that the external models provide the inlet and outlet effort to the resistive element, thus allowing the resistive element to calculate the flow from these efforts and provide that data to the connected models.

As the development of the DTMS Framework progressed, it became apparent that Equation 2.6 was much too limiting for the types of components that would be modeled using the DTMS Framework, and thus it became necessary to add generality to this equation to allow a wider range of models to be simulated. As shown in Section 2.1.1, the more general Equation 2.1 was adopted in order to allow the flow to be calculated from any function of the inlet and outlet efforts, rather than restricting it to the difference between these efforts. However, the single-inlet/single-outlet condition remains a requirement for flow-based models, and thus the generic flow equation presented in Equation 2.5 is currently used as the basis for flow-based components in the DTMS Framework. The bond graph equivalent of this component remains similar to the diagram presented in Figure 2.3, but removes the 1-junction to allow the two efforts to flow directly into the resistive element:

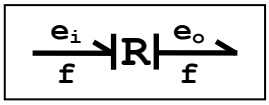


Figure 2.4: Bond graph equivalent of current DTMS flow-based component

2.2.3 Effort-Based Components

Representing effort-based components using bond graph elements is not as easy as it was for flow-based components, since there is no single bond graph element that precisely corresponds to the behavior of the effort-based component. Returning to Section 2.1.2, effort-based components are defined by the flow-conservation equation presented in Equation 2.3 and reproduced here:

$$\sum_{j=1}^{m} f_{i,j} - \sum_{k=1}^{n} f_{o,k} = 0 \tag{2.7}$$

Conservation equations are generally modeled in bond graph theory using either a 1-junction or a 0-junction; in this case, a 0-junction provides conservation of flow and thus is appropriate for use. However, one of the primary roles of the effort-based component is to allow flow-based models to be connected together. If a 0-junction is the only model used to represent an effort-based component, the following bond graph is produced when connecting together two flow-based components:

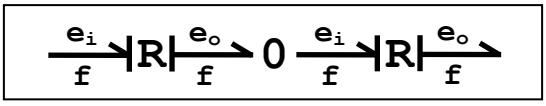


Figure 2.5: Bond graph using a 0-junction to represent an effort-based model

This diagram is invalid based on the causality rules for bond graphs, which require that exactly one branch of a 0-junction provide the effort to the junction itself, which in turn is

passed to other models connected to the 0-junction. In this diagram, neither flow-based model is capable of providing an effort to the 0-junction, therefore preventing this diagram from representing a valid bond graph system.

Another model must be added to the 0-junction that will provide it with the effort that is needed to make the system valid. While the result may seem somewhat surprising, the desired behavior can be achieved by using the following bond graph structure to represent an effort-based component:

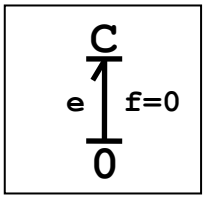


Figure 2.6: Bond graph equivalent of an effort-based model

Figure 2.6 demonstrates a capacitive model that does not allow any flow to pass through it, or in other words, it represents a capacitor with no capacitance. Physically, this structure is virtually meaningless in any system, since it has no impact on the flow network, but from a modeling perspective it allows the efforts of the system to become the independent variables that are used to describe system behavior. Each of the flow-based models provides the system with a flow equation that is dependent upon the values of the efforts, and the flows are all related based on the conservation-of-flow equations present at each effort-based model. By combining these, a system of algebraic equations can be produced that contain the efforts of the system as the only unknown variables. Finding the solution for the system can then be accomplished by merely solving this system of equations for the efforts of the modeled system. Further details concerning the

construction of this system of equations will be presented in section 2.3.4, and a detailed explanation of the solution methods can be found in the thesis of Patrick Paullus [20].

2.2.4 Source Components

Source components in the DTMS Framework can be directly represented in bond graph theory using the bond graph source elements for flow and effort. Since the flow source components must have at least one connection but could have both an input and an output connection, these components can be represented in bond graph theory using either of the structures found in Figure 2.7. The flow direction has been intentionally removed from these structures since all configurations are possible; i.e., the flow can pass into, out of, or through a flow source component:

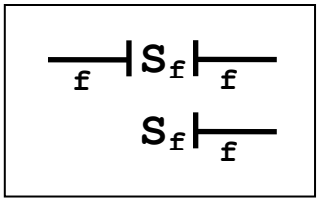


Figure 2.7: Two bond graph equivalents of DTMS flow-source component

The effort source component is represented using the bond graph effort-source element found in Figure 2.8, which has been constructed to resemble the effort-based component. Again, the flow direction has been intentionally removed from this structure since all flow configurations are possible.

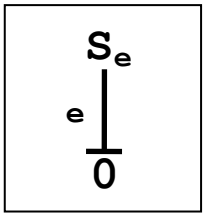


Figure 2.8: Bond graph equivalent of an effort-based model

2.3 BENEFITS OF RESISTIVE NETWORK MODELING

The details of the components that make up the Resistive Network modeling strategy alone do not fully illustrate the many benefits that are provided by this modeling approach. In order to achieve the intended usefulness and general applicability of the DTMS Framework, the modeling strategy chosen for any system must adhere to certain design goals that will allow the framework to become an integral part of the design process for the Navy's All-Electric Ship. Resistive Network modeling was specifically chosen for the DTMS Framework due to its applicability to numerous energy domains, component-independent modeling philosophy, multiple layers of detail, and straightforward solution methods. These benefits are discussed in the following sections.

2.3.1 Multiple Energy Domain Applicability

While the Resistive Network modeling strategy is most often described using the electrical circuit analogy, the general principles of power transmission using the generic parameters of effort and flow are applicable to virtually any type of physical system. As long as the physical parameters representing effort and flow are chosen appropriately, the Resistive Network modeling strategy can be used to simulate translational, rotational, mechanical, electrical, electro-mechanical, hydraulic, and many other types of systems. Furthermore, the generic nature of power transmission allows multiple types of energy systems to be combined together within a single modeling environment. This is particularly useful in application to an All-Electric Ship where a myriad of energy systems are working in sequence, including: compressible fluid systems driving the prime movers, rotational shaft networks transferring power from the prime movers to the electrical generators, electro-mechanical generators that transfer power from the rotational shafts to the electrical system, an electrical grid which distributes electrical

power for electronic devices on board the ship, and a series of fluid systems that provide cooling flow to various ship systems.

2.3.2 Component-Independent Development

In a distributed modeling environment where various models are created and utilized by numerous individuals for a variety of purposes, it becomes crucially important to minimize the interdependencies between different groups of models in order to allow them the broadest possible applicability. Modeling systems exist to ease the development process for simulation users; therefore reducing the amount of work required by the users is an essential design focus.

The principles of power transmission utilized by the Resistive Network modeling strategy allow each component of a system to be modeled completely independently from the models surrounding it and from the system being simulated. Individual models in resistive networks are able to completely calculate their state using data provided by the bounding models. Primarily, flow models only require the inlet and outlet effort values from the bounding models, although additional domain-dependent variables (e.g., thermal properties) may also be incorporated from an upstream model.

This freedom allows model developers to completely design, implement, and test each and every resistive network model in isolation, without concern for any external system limitations or specializations.

2.3.3 Multiple Levels of Detail

An additional benefit provided by virtue of complete model independence offered by the Resistive Network modeling strategy is the ability to utilize differing levels of model complexity within various parts of a single simulation. Since each model is designed separately, certain crucial sections of a system may implement highly detailed

models that provide very accurate simulation, while other less crucial sections may utilize more generalized models as appropriate. The model developer decides exactly how detailed a particular simulation must be based on the complexity of the models that are utilized, and since different levels of complexity can be seamlessly integrated into a single simulation, the developer can focus on the most important data for the task at hand while simultaneously minimizing the simulation time used to calculate less important aspects in a system simulation.

2.3.4 Straightforward Solution Methods

The generic models that compose the Resistive Network modeling strategy were designed specifically to construct the overall flow network in a very predictable and repeatable manner. This consistent nature allows a system to be addressed using generic methods that need not be specialized for each individual simulation.

In bond graph theory, the number of independent variables in a system is defined by the presence of capacitive and inertial elements and the causality associated with each. As addressed in Section 2.2, the only independent variables located in resistive networks are the capacitive models found in effort components. Since these models contain zero flow through them, the only important parameter to their operation is their effort. Therefore, these are the only parameters that must be resolved in order to properly simulate a resistive network system.

Thanks to this behavior, the entire set of system equations can be reduced to a single matrix equation, which can be solved using a variety of linear algebra techniques. The potentially nonlinear nature of the flow equations often makes this system of equations quite complicated, but a variety of iterative methods can be introduced to quickly and accurately reveal the solution.

2.4 FLUID FLOW EXAMPLE

At first, the Resistive Network modeling strategy may appear to be overly restrictive in terms of its applicability for the modeling of diverse physical systems. The construction of a flow equation based only on the inlet and outlet efforts may seem unnatural or even impossible based on the known physical equations for a particular model. However, once the user becomes familiar with the Resistive Network modeling strategy, the process becomes straightforward, can be implemented with ease, and reveals itself as widely applicable to numerous systems. The following example constructs the flow equation for a generic fluid flow component to demonstrate how this modeling strategy is applied to a typical and common physical system.

The Resistive Network modeling strategy is predicated on two basic assumptions pertaining to the component that is being modeled: *energy conservation* and *quasi-steady flow*. In this context, the term "steady flow" refers only to the flow variable of the current system, and should not be confused with the more restrictive term commonly used for fluid systems that requires that mass, volume, and total energy content remain steady throughout the system. Beginning with energy conservation, the following equation forms the basis for this fluid flow example:

$$\Delta E_{total} = 0 \tag{2.8}$$

where ΔE_{total} represents the total energy within the component at any point in time. In general, the total amount of energy can be separated into the net energy crossing the boundaries of the component and the net energy stored in the component:

$$\Delta E_{boundary} + \Delta E_{stored} = 0 \tag{2.9}$$

However, since the Resistive Network modeling strategy also requires steady flow, there cannot be any change in the amount of energy stored within the component. Therefore:

$$\Delta E_{stored} = 0 \tag{2.10}$$

which reduces Equation 2.9 to:

$$\Delta E_{boundary} = 0 \tag{2.11}$$

From this point on, the subscript *boundary* is assumed for all difference parameters and thus will be dropped from all further equations.

For fluid flow, the energy transferred across the component boundaries can be represented using the following equation:

$$\Delta U + \Delta PE + \Delta KE = \delta Q - \delta W \tag{2.12}$$

where ΔU is the difference in the internal energy of the flows across the boundaries, ΔPE is the difference in the potential energy of the flows across the boundaries, ΔKE is the difference in the kinetic energy of the flows across the boundaries, δQ is net heat added to the component, and δW is the net work performed by the component. Each of the delta parameters in Equation 2.12 is measured as the energy of an outlet flow minus the energy of an inlet flow.

As is typically done, all forms of potential energy excluding gravitational potential energy will be neglected for the large-scale simulation of a stable, non-reacting, electrically-neutral, low-viscosity Newtonian fluid. Thus, the potential energy can be reduced to the following:

$$\Delta PE = \Delta(mgz) \tag{2.13}$$

where m is the mass of the fluid, g is the gravitational constant, and z is the height of the fluid at either the inlet or outlet. Furthermore, the kinetic energy is assumed to be associated with translational motion and can be replaced by the following:

$$\Delta KE = \frac{\Delta(mv^2)}{2} \tag{2.14}$$

where m is again the mass of the fluid and v is the mean, uniform velocity of the fluid across either an inlet or outlet. Incorporating Equations 2.13 and 2.14 into Equation 2.12:

$$\Delta U + \Delta(mgz) + \frac{\Delta(mv^2)}{2} = \delta Q - \delta W \tag{2.15}$$

In addition to requiring that there can be no change in the amount of energy stored within the component, steady flow for fluid systems also requires that there can be no change in the amount of mass present within each component. The usage of "steady flow" in this context is different from the common usage in fluid dynamics. While steady flow in fluid dynamics would require that the mass and volume remain steady across the inlet and outlet boundaries, steady flow in the Resistive Network modeling strategy only requires that the flow variable for the current energy domain remain steady. In compressible fluid systems, the mass flow rate is used to represent the flow characteristics, and thus only the mass must remain steady across the boundaries of each flow model, which is reflected in the following equation. It is also assumed that the model size is sufficiently small such that gravitational effects are similar at the inlet and outlet.

$$\Delta U + mg\Delta z + \frac{m\Delta\left(v^2\right)}{2} = \delta Q - \delta W \tag{2.16}$$

The net work performed on the component can be defined as the work performed on the boundaries, plus any external work that is introduced to or extracted from the fluid, which is here restricted to "shaft work":

$$\delta W = \Delta\left(PV\right) + \delta W_{shaft} \tag{2.17}$$

where P is the pressure and V is the volume of the fluid at the boundary. Changes in internal energy can be characterized by the heat transfer into the component plus any energy that was dissipated through unmeasured means such as friction, ambient heat transfer, etc.:

$$\Delta U = \delta Q + E_{lost} \tag{2.18}$$

where E_{lost} is the dissipated energy. While this energy is never truly "lost" in the conventional sense, it represents the transfer of energy from the component that is not encompassed by the model. Combining these two equations with Equation 2.16 results in the following:

$$\Delta\left(PV\right) + mg\Delta z + \frac{m\Delta\left(v^2\right)}{2} + \delta W_{shaft} + E_{lost} = 0 \tag{2.19}$$

By solving for Δv^2 and expanding the delta terms, the following equation is generated:

$$\Delta v^2 = -2\left(P_{out}\frac{V_{out}}{m} - P_{in}\frac{V_{in}}{m} + g\Delta z + \delta w_{shaft} + e_{lost} \right) \tag{2.20}$$

where P_{out} and V_{out} are the pressure and volume at the outlet respectively, P_{in} and V_{in} are the pressure and volume at the inlet respectively, δw_{shaft} is the shaft work per unit mass, and e_{lost} is the specific dissipation energy. The following definitions for density and mass flow rate can then be applied:

$$\frac{V}{m} = \frac{1}{\rho} \tag{2.21}$$

$$v = \frac{\dot{m}}{\rho A} \tag{2.22}$$

where V is the fluid volume, m is the fluid mass, ρ is the fluid density, v is the fluid velocity, $\dot{m}$ is the mass flow rate of the fluid, and A is the cross-sectional area for flow at a particular point. Substituting these definitions into Equation 2.20:

$$\left(\frac{\dot{m}}{\rho_{out} A_{out}}\right)^2 - \left(\frac{\dot{m}}{\rho_{in} A_{in}}\right)^2 = -2 \cdot \left(\frac{P_{out}}{\rho_{out}} - \frac{P_{in}}{\rho_{in}} + g\Delta z + \delta w_{shaft} + e_{lost}\right) \tag{2.23}$$

This equation can be solved for the mass flow rate:

$$\dot{m} = \sqrt{\frac{2}{\left(\frac{1}{\rho_{in} A_{in}}\right)^2 - \left(\frac{1}{\rho_{out} A_{out}}\right)^2}} \cdot \sqrt{\frac{P_{out}}{\rho_{out}} - \frac{P_{in}}{\rho_{in}} + g\Delta z + \delta w_{shaft} + e_{lost}} \tag{2.24}$$

Equation 2.24 represents the basic flow equation for compressible fluid flow through a component, which can be generalized into the following:

$$f = C \cdot \sqrt{\Delta e + S} \tag{2.25}$$

In this equation, the flow f is a function of a conductance term C, which is multiplied by the square root of the difference between the inlet and outlet efforts Δe that is added to an effort source term S. This source term represents any internal mechanism which contributes to a change in the effort as flow passes through the component. For the compressible fluid flow equation, the general flow terms are represented by the following:

$$f = \dot{m} \tag{2.26}$$

$$e = \frac{P}{\rho} \tag{2.27}$$

$$C = \sqrt{\frac{2}{\left(\frac{1}{\rho_{in} A_{in}}\right)^2 - \left(\frac{1}{\rho_{out} A_{out}}\right)^2}} \tag{2.28}$$

$$S = g\Delta z + \delta w_{shaft} + e_{lost} \tag{2.29}$$

Equation 2.24 can be extended to incompressible fluid flow by applying the assumption that the density is constant, and thus $\rho = \rho_{in} = \rho_{out}$:

$$\dot{m} = \sqrt{\frac{2\rho^2}{\frac{1}{A_{in}{}^2} - \frac{1}{A_{out}{}^2}}} \cdot \sqrt{\frac{\Delta P}{\rho} + g\Delta z + \delta w_{shaft} + e_{lost}} \tag{2.30}$$

In incompressible flow, the volumetric flow rate is linearly proportional to the mass flow rate, so Equation 2.30 can be solved for the volumetric flow rate:

$$Q = \sqrt{\frac{2}{\frac{\rho}{A_{in}{}^2} - \frac{\rho}{A_{out}{}^2}}} \cdot \sqrt{\Delta P + \rho(g\Delta z + \delta w_{shaft} + e_{lost})} \tag{2.31}$$

This equation also fits the general form of the flow equation presented in Equation 2.25 with the general terms representing the following:

$$f = Q \tag{2.32}$$

$$e = P \tag{2.33}$$

$$C = \sqrt{\frac{2}{\frac{\rho}{A_{in}{}^2} - \frac{\rho}{A_{out}{}^2}}} \tag{2.34}$$

$$S = \rho\left(g\Delta z + \delta w_{shaft} + e_{lost}\right) \tag{2.35}$$

Throughout the DTMS Framework, Equation 2.25 is used as the basis for many of the fluid flow components that have been modeled. Starting with basic domain equations and conservation laws, the equations for nearly any independent model can be converted into a form that is compatible with the Resistive Network modeling strategy and thus can be used within the DTMS Framework. Further examples of constructing electrical flow equations for use with the Resistive Network modeling strategy can be found in the thesis of Matthew Pruske [23].

2.5 Comparison to Resistive Companion Modeling

As an alternative to the Resistive Network modeling strategy, the Resistive Companion modeling strategy is a common approach which has been used in other simulation environments that are similar to the DTMS Framework. Although sharing many characteristics, Resistive Network modeling provides several distinct benefits over

the Resistive Companion modeling, specifically direct solving of nonlinear equations and flexibility in solution methods. This section will address the similarities and differences between the two strategies and focus on the advantages that the DTMS Framework gains by using Resistive Network modeling as its primary modeling method.

2.5.1 Resistive Companion Equation

At its core, Resistive Companion modeling is based on the same modeling principles as the Resistive Network modeling strategy. Both methods are based on the principles of mass and energy conservation and steady flow, and each strategy uses the transmission of power as the connective link between independently developed models. As their names imply, both methods involve the modeling of physical systems as resistance networks like those that were discussed in Section 2.1. However, the primary difference between the two strategies is in the flow equation that forms the basis for all models that are implemented in each system.

In Resistive Companion modeling, the flow that is calculated through each component of a system must conform to the following general equation:

$$I(t) = G(t) \cdot V(t) - b(t - h) \tag{2.36}$$

In this equation, the following general terms are treated as functions of time: *I(t)* represents the flow at the current time point, *G(t)* is a conductance term calculated at the current time point, *V(t)* represents the difference between the inlet and outlet efforts at the current time point, and *b(t – h)* is an effort source term calculated at the previous time point, with *h* representing the current time step in the simulation.

2.5.2 Nonlinear Flow Equations

The resistive companion equation shown in Equation 2.36 requires a linear relationship between the flow and the effort differences across a flow component. While this relationship is satisfactory for most simplistic simulations, it is inadequate for advanced physical systems. Numerous physical components have complex nonlinear relationships between flow and effort and thus cannot directly utilize the resistive companion modeling strategy in the form presented.

To address this, a time-based approximation of the nonlinear flow equation is constructed by employing the concept of linearization to produce an equation that fits the desired form. Linearization depends upon an approximation of the derivative of the flow with respect to the effort difference at two consecutive time steps:

$$\frac{dI}{dV} \approx \frac{\Delta I}{\Delta V} = \frac{I(t) - I(t-h)}{V(t) - V(t-h)} \tag{2.37}$$

This equation is then solved for *I(t)*:

$$I(t) = \frac{dI}{dV} \cdot V(t) - \left[\frac{dI}{dV} \cdot V(t-h) - I(t-h) \right] \tag{2.38}$$

Equation 2.38 now fits the required form of the resistive companion equation by recognizing the following identities:

$$G(V) = \frac{dI}{dV} \tag{2.39}$$

$$b(t-h) = \frac{dI}{dV} \cdot V(t-h) - I(t-h) \tag{2.40}$$

The equations for G and b are now both functions of the effort difference V, and thus it becomes necessary to perform an iteration in order to converge upon the correct solution, which is most commonly achieved by utilizing the Newton-Raphson method.

2.5.3 Disadvantages of Resistive Companion Modeling

As demonstrated in Section 2.2.2, the primary flow equation for the Resistive Network modeling strategy shown in Equation 2.5 is far more general than that presented by the resistive companion method in Equation 2.36. In Resistive Network modeling, the flow can vary based on any function of the inlet and outlet efforts. The Resistive Companion modeling strategy requires that flow models vary specifically with respect to the *difference* between the inlet and outlet efforts. In certain models, particularly ones involving compressible fluids, the primary flow varies only with respect to the inlet effort, as shown in the following flow equation for a simple axial compressor:

$$\dot{m} = K_1 \cdot \frac{P_i}{T_i} \cdot N \tag{2.41}$$

with $\dot{m}$ representing the mass flow rate, P_i representing the inlet pressure, T_i representing the inlet temperature, N representing the rotational speed of the compressor, and K_1 representing a system-dependent design constant obtained through parameterization. In this model, the mass flow rate represents the flow parameter, and the pressure represents the effort parameter. Since this equation does not contain any reference to the difference between the inlet and outlet efforts, it is impossible for this to be expressed using the resistive companion equation. Due to the more general flow equation used by the Resistive Network modeling strategy, models with this type of flow equation and many other variations can be and have been implemented in the DTMS Framework without requiring any modification or approximation.

As addressed in the previous section, models with nonlinear flow equations must be approximated when used with the Resistive Companion modeling strategy through linearization techniques. Although the resulting linearized equation will converge to the proper steady state solution, the behavior of the model is now time dependent, and its behavior in dynamic scenarios can vary depending upon the time step that is used when simulating the system. Specifically, models and simulated systems that rely heavily on these linearized equations respond poorly to step changes in the boundary conditions of the system. The apparent solution to this limitation of the Resistive Companion modeling strategy is to increase the complexity of the boundary sources, thus modeling the step change behavior as a series of smaller step changes spread out over a larger number of time steps. However, this solution directly violates the principles of complete model independence by tying the accuracy and performance of a specific model to the overall complexity of the system in which it is being used. By allowing the direct use of nonlinear flow equations within models, the Resistive Network modeling strategy does not suffer from this limitation, and the physics of a model can be accurately implemented entirely within the model itself, without requiring any additional complexity to be added to the boundary sources or surrounding models.

While the use of a simplified linear flow equation suffers from limitations of applicability, disadvantages in dynamic accuracy, and incompatibility with the principles of independent modeling, it does have the advantage of requiring simpler numerical methods for the solution of complex systems. A system of Resistive Companion equations can be reduced to a single matrix equation that can be solved directly at each time step. When nonlinear flow approximations are utilized, this matrix equation can be solved iteratively using convergence techniques such as the Newton-Raphson method. Resistive Network modeling requires a more complex set of solution methods, but it

allows greater freedom in the selection of the most appropriate method based on the system to be simulated. Within the DTMS Framework, three solution techniques have been implemented which allow for varying levels of accuracy and speed. In the current construct, primarily linear systems can take advantage of the speed improvements provided by linearization techniques, highly nonlinear stable systems can utilize standard Newton-Raphson methods for quick and accurate solutions, and highly nonlinear less-stable systems can utilize techniques with a higher likelihood of convergence. More details about the various solution methods currently provided with the DTMS Framework can be found in the thesis of Patrick Paullus [20]. Significantly, the model developer is unconstrained and free to use any of these techniques, or introduce other approaches, as deemed necessary and appropriate by the developer.

2.6 CHAPTER SUMMARY

Although it may require more complex solution methods, the Resistive Network modeling strategy provides numerous advantages over the similar, but more limited, modeling strategy provided by the Resistive Companion approach. With increased generality and applicability to a broader range of physical components, advantages in overall system accuracy, and increased flexibility in solution methodology, the Resistive Network modeling strategy was chosen as the primary modeling system for use in the DTMS Framework. The open and adaptive nature of the framework allows numerous alternative modeling strategies to be implemented and utilized within the DTMS environment. The overall construction of the DTMS Framework and the integration of the Resistive Network modeling strategy are discussed further in the next chapter.

Chapter 3: Dynamic Thermal Modeling and Simulation Framework

When the ESRDC was formed in 2002, the work being performed by the various universities in the consortium required advanced modeling techniques to properly simulate the various technology approaches for the Navy's All-Electric Ship program. At the outset of the consortium's work, few modeling tools were available to specifically address the Navy's diverse requirements, which include high-energy power generation systems, robust electrical grids with dynamic reconfigurability, high-density electrical storage capabilities, intense pulse-load systems including advanced radar, sonar, and weaponry, and complex adaptive thermal management systems. With such widespread and unique challenges facing the members of the consortium, each university turned to developmental and commercial software systems to address their specific modeling needs.

Within the thermal management team at the University of Texas at Austin, research efforts focused on the use of two primary modeling tools in order to address the simulation of thermal aspects of both current and near-term naval technologies. Initially steady-state representations were primarily accomplished using a thermal modeling tool created by the Delft University of Technology (Netherlands) called *CycleTempo*, which was designed to address thermodynamic analysis and optimization of systems for the production of electricity, heat, and refrigeration. Various detailed thermodynamic representations were developed at UT using this tool including a hybrid gas-turbine engine [14], a solid oxide fuel cell [3], and a 200-ton chiller model [10]. Dynamic simulations were performed using a commercial power plant modeling tool called *ProTRAX* that was developed over many years by TRAX International. Utilizing an adaptable FORTRAN programming base and a flow-effort-based modeling system

similar to the DTMS Framework, indefinitely large thermodynamic systems could be modeled in the *ProTRAX* environment. This environment also featured an integrated system of robust, tunable feedback controls allowing for the generation of advanced and sophisticated simulation results. Dynamic models of shipboard pulsed weaponry [29], the marine 200-ton chiller [10], and a notional Integrated Electric Propulsion System [14] were created by UT graduate students using the *ProTRAX* modeling environment.

While these modeling tools have proven immensely useful in our research efforts aimed at the simulation of thermal management systems for the All-Electric Ship, all commercial software packages, in hindsight, have significant drawbacks that limit their suitability as a primary modeling tool for the US Navy. Specifically, the research vision for the ESRDC includes development of accessible modeling tools capable of capturing the dynamic interactions between the various mechanical, electrical, and thermal systems onboard future naval ships. The commercial software packages initially utilized by the consortium members lack this universal approach to diverse system modeling and are often tailored for applications other than shipboard systems. Coupled with high investment costs, general lack of portability, and minimal customization options, recent research focus within the consortium has leaned away from commercial software toward development of in-house simulation environments that are easily transportable to other users.

In December of 2007, UT graduate student Patrick Paullus completed initial development work on an in-house code, which was later named the Dynamic Thermal Modeling and Simulation (DTMS) Framework, to specifically address the thermal management modeling needs at the University of Texas [20]. Designed from the ground up to be a highly-customizable and universally-applicable analysis tool, the DTMS Framework has become the primary development system for all of the ESRDC thermal

management work currently performed at the University of Texas. Several recent research efforts have been completed that utilize the power of the DTMS Framework, which include a simulation of the entire starboard freshwater chilling loop of the DDG-51 guided missile destroyer with advanced chiller technologies and dynamic control systems [13], and a thermal-electrical co-simulation of the electrical zonal distribution system that makes up the Naval Combat Survivability (NCS) Testbed [23].

Over the past two years, the DTMS Framework has grown and advanced into a much more sophisticated software platform capable of modeling and simulating complex physical systems while extending the ease of use for both model developers and simulation users. The remaining chapters of this thesis describe the various advances that have been incorporated into the DTMS Framework that have allowed it to become the primary tool for thermal management research within the ESRDC.

This chapter focuses on the architecture and implementation of the DTMS Framework, including key object-oriented programming concepts, the basic class hierarchy that makes up the modeling interface of the simulation system, and various development pieces that produce a complete DTMS simulation.

3.1 C++ OBJECT-ORIENTED PROGRAMMING CONCEPTS

Before providing a hierarchical overview of the DTMS Framework, it is beneficial to address several of the object-oriented programming concepts that contribute heavily to the overall design of the DTMS Framework and result in its highly adaptive nature. These concepts form the backbone of the design methodology utilized within the framework and are important considerations for future model developers.

Before the mid-1990's, most mainstream software application development involved programming languages based on the principles of the *procedural programming* paradigm. Using this approach, application code could be separated into sets of functions

and subroutines (collectively called *procedures*) that allowed the simplified reuse of common coding routines with differing input sets. Common languages to utilize this programming paradigm are C, FORTRAN, and Pascal.

However, as programming tasks have grown exponentially in both complexity and scope, software developers have developed more sophisticated programming systems in order to both reduce code maintenance costs and promote overall system quality. Based on these needs, *object-oriented programming* has evolved to shift the general design focus from the *behavior* of a code module to the *data* (or *objects*) contained within the module. Object-oriented programs should be viewed as a collection of interacting data objects, rather than a series of tasks to be completed. While initially developed much earlier, this programming paradigm did not enjoy widespread use among software applications until the mid-1990's, yet it remains one of the most popular paradigms in use today as demonstrated by now common programming languages such as C++, C#, Objective-C, and Java.

3.1.1 C++ Classes

The design principles of object-oriented programming are based on the concept of a *class*, which represents a self-sufficient collection of data containing all of the functionality needed to properly manipulate that data. In C++, classes represent a template for a new type of data that declares the variables and functions that will be used to define new data objects, which is demonstrated through the following example.

A calendar date can be considered as a collection of three integers which convey a very specific meaning when used together. These numbers are also bounded and interrelated: months range from 1 through 12 and days range from 1 to either 28, 29, 30, or 31 depending on the value of the month and year. In procedural programming, it can be very difficult and costly for program developers to consistently enforce these rules

throughout all of the possible uses of the calendar date structure. However, object-oriented programming allows the variables that represent the month, day, and year to be collected together with the rule-enforcement routines to easily guarantee the validity of these data members.

A possible C++ class definition for the calendar date is presented in Figure 3.1:

```
class calendarDate
{
private:
      int month_;
      int day_;
      int year_;

public:
      calendarDate()
      {
            //Assign default values for the internal variables
            month_ = 1;
            day_ = 1;
            year_ = 1970;
      }

      void setDate(int month, int day, int year)
      {
            /*
            Code to enforce rules, such as forcing all values
            to be positive, limiting months to values between
            1 and 12, and requiring the correct number of days
            in each month.  If the given values do not meet
            the requirements, they can be given default values
            or an error can be reported.
            */
      }

      //Allow read-only access to the internal variables
      int getMonth() { return month_; }
      int getDay()   { return day_; }
      int getYear()  { return year_; }
};
```

Figure 3.1: Sample class definition for a calendar date

In this example, it is important to note that various aspects of a C++ class can have different *access specifiers*. In this case, the variables that contain the values for the month, day, and year are given *private* access, indicating that only functions contained within the class may read or modify their values. The member functions associated with the class are given *public* access which allows them to be used by functions that exist outside the class definition. A third type of access specifier known as *protected* access allows variables or functions to be used only by functions contained within the class or functions contained within classes that are derived from this class. Inheritance and derived classes are discussed in Section 3.1.3.

The first function shown in the `calendarDate` class definition has the same name as the class itself and is called the *constructor*. This function is automatically called whenever a `calendarDate` class is created and is commonly used to initialize the data members of the class. For the `calendarDate` class, the constructor is used to assign the default date of January 1, 1970 to the month, day, and year variables. This is the first rule-enforcement function to ensure that all `calendarDate` objects have a valid date, even before the user has explicitly provided one.

The next function, `setDate`, is the only means by which the internal values of the month, day, and year variables can be modified by an outside user. While the detailed rule-enforcement code was removed from Figure 3.1 for simplicity, this function is used to ensure that the date provided by the user conforms to the known requirements. As mentioned in the code comment, any invalid dates can be given default values or an error can be reported to the user.

The final three functions `getMonth`, `getDay`, and `getYear` each allow external read-only access to the internal variables that represent the month, day, and year. These allow external functions to acknowledge and act upon the values of the `calendarDate`

object, but do not allow the manipulation of these values except through the `setDate` function where the context-specific rule set can be enforced.

The concepts of access control and rule enforcement are paramount to the object-oriented programming paradigm and are widely used throughout the DTMS Framework. However, these are not the only benefits provided by this programming methodology, and additional benefits are explored in the following sections.

3.1.2 Class Containment

In addition to containing the basic data types provided by the C++ programming language, such as integers, characters, and floating-point numbers, classes can contain the objects created from other class definitions. The calendar date example presented in the previous section can be utilized in a new class definition representing an appointment book that contains a list of appointments and the dates on which they occur. A possible C++ class definition for this class is presented in Figure 3.2:

```
class appointmentBook
{
private:
      //Arrays to hold the names and dates of the appointments
      string appointmentNames[100];
      calendarDate appointmentDates[100];
      //Integer variable holds current number of appointments
      int numberOfAppointments;
      //Internal function to sort appointments by date
      void sortAppointmentsByDate();
public:
      //Constructor
      appointmentBook();
      //External function to add an appointment
      void addAppointment(string name, calendarDate date);
      //Read-only functions to allow access to the name and
      //date of the next appointment
      string getNameofNextAppointment();
      calendarDate getDateofNextAppointment();
};
```

Figure 3.2: Sample class definition for appointment book

Containment in an object-oriented programming language promotes extended code reuse and reduced maintenance requirements throughout the growth in program complexity. In the example presented above, the appointment book gains the benefits of date rule-enforcement simply by including the calendar date class created earlier. By separating a complex system into a series of specialized components, detailed and sophisticated programming tasks can be accomplished with reduced development effort.

3.1.3 Class Inheritance

While containment allows for components of a class to gain the benefits and behaviors of the variables that are contained, inheritance allows the entire class to make use of the behaviors and variables in the inherited class and allows the derived class to supplement and even modify those behaviors to better suit the nature of the class. With inheritance, a new class definition, called the *derived class*, is created from an existing definition, called the *base class*. The new class will contain all of the variables and functions of the base class that were given either the *public* or *protected* access specifier. Inheritance can also add additional variables and functionality to the existing code base.

Figure 3.3 shows an example of a base class containing information about a type of literary publication:

```
class publication
{
protected:
      string author_;
      calendarDate publicationDate_;

public:
      publication(string author, calendarDate date);
};
```

Figure 3.3: Sample class for publication base class

Any classes that are derived from this publication base class will contain the author and publication date variables, along with any additional functions that might be added to the publication base class in the future. The derived classes can add additional data as presented in Figure 3.4:

```
class book : public literaryPublication
{
protected:
      string publishingCompany_;
      string classificationNumber_;

public:
      book(string author, calendarDate date,
            string company, string classificationNumber);
};

class journalArticle : public literaryPublication
{
protected:
      string journalName_;
      int volumeNumber_;
      int issueNumber_;

public:
      journalArticle(string author, calendarDate date,
            string journalName, int volumeNumber
            int issueNumber);
};
```

Figure 3.4: Sample class for publication derived classes

At its most basic level, inheritance is used in object-oriented languages to promote code reuse, as was demonstrated by the previous example. Neither the `book` class nor the `journalArticle` class needs to create its own variables for the author or publication date, since these are already handled by the base class `publication`. More importantly, the above example demonstrates the use of class inheritance to introduce increasing specialization throughout a series of derived class that each introduces a manageable level of complexity. The DTMS Framework relies heavily on the feature of

inheritance; for example, a generic class has been created to represent any type of model in DTMS, which is then used to create specializations for the flow and effort models found in the Resistive Network modeling strategy. Further specializations are made for the various energy domains (fluid, thermal, electrical, etc.), and finally specific models have been created to represent physical components such as pipes, fans, turbines, etc. Each derived class may focus solely on the functionality that it adds to the current design, but the final product will incorporate all of the functionality provided in the class hierarchy. Class diagrams that present the inheritance relationships for class of the DTMS Framework in a graphical format are provided in Appendix B.

3.1.4 Polymorphism

The most useful feature of inheritance, polymorphism, is a feature that allows derived classes to act as substitutes for their base classes, since they contain exactly the same information as the base classes, but still take advantage of any overridden functionality in the derived class. It leads to the immense power that object-oriented programming provides. With this feature, existing code can automatically take advantage of any newly-created derived classes without requiring any knowledge of the details of the derived classes themselves.

The following set of figures demonstrates the power of polymorphism:

```
class Shape
{
public:
      virtual double calculateArea() = 0;
      virtual double calculatePerimeter = 0;
};
```

Figure 3.5: Polymorphic base class for shapes

In this base class, two functions are supplied, `calculateArea` and `calculatePerimeter`, which provide the functionality implied by their names in the derived classes presented in Figure 3.6. Specific calculations for area and perimeter vary widely across different geometric structures, and thus there are no generic calculations that can be performed for these functions. The generic `Shape` class cannot provide code for these two functions, so it declares that the derived class must provide this functionality by placing the keyword `virtual` in front of the function declaration and the "`= 0`" syntax after it. These two functions are referred to as *pure virtual functions*, which allow the derived class behavior to be maintained when a derived class object is substituted for a base class object.

Figures 3.6 and 3.7 show several examples of classes derived from the `Shape` class, which override the `calculateArea` and `calculatePerimeter` functions to provide customized behavior:

```
class Circle : public Shape
{
protected:
      double diameter_;

public:
      Circle(double diameter)
      {
            diameter_ = diameter;
      }

      virtual double calculateArea()
      {
            return ((PI / 4) * diameter_ * diameter_);
      }

      virtual double calculatePerimeter()
      {
            return (PI * diameter_);
      }
};
```

Figure 3.6: Polymorphic derived class for circles

```
class Rectangle : public Shape
{
protected:
	double length_;
	double width_;

public:
	Rectangle(double length, double width)
	{
		length_ = length;
		width_ = width;
	}

	virtual double calculateArea()
	{
		return (length_ * width_);
	}

	virtual double calculatePerimeter()
	{
		return (2 * (length_ + width_));
	}
};
```

Figure 3.7: Polymorphic derived class for rectangles

Each of these classes provides the data needed to properly calculate the area and perimeter of the respective shape. Generic functionality can now be created which takes advantage of this polymorphic behavior, based solely on the definition of the `Shape` base class, as shown in the following figure:

```
void printData(Shape & currentShape)
{
	cout << "Area: ";
	cout << currentShape.calculateArea() << endl;
	cout << "Perimeter: "
	cout << currentShape.calculatePerimeter() << endl;
}
```

Figure 3.8: Function utilizing polymorphic behavior of the `Shape` class

The usage of the `printData` function is demonstrated in Figure 3.9:

```
int main()
{
      Circle myCircle(4.0);
      printData(myCircle);

      Rectangle myRectangle(5.3, 6.2);
      printData(myRectangle);

      return 0;
}
```

Figure 3.9: Usage of polymorphic functionality

In this example, the `printData` function will display the area and perimeter for any type of `Shape` class, regardless of the specific implementation in the derived class. Most notably, this function can take advantage of any class derived from `Shape`, including any potential future classes which have not yet been created. If, for example, a developer decides that a `Triangle` class is needed, this new class can take advantage of the functionality provided by the `printData` function by simply implementing the `calculateArea` and `calculatePerimeter` functions properly. No changes need to be made to the `printData` function in order to extend to this newly-created class.

Polymorphism is the enabling technology that allows the DTMS Framework to extend to infinitely large systems of customized component behavior. Using a sophisticated inheritance hierarchy, the various aspects of the framework can take advantage of numerous diverse and complex physics-based models without requiring that any changes be made to the underlying system. Specifically, any type of model, control, fluid, or solver can be added to the simulation system, and any customized behavior built into those specific components will be captured through the generalized simulation architecture.

3.2 STRUCTURE OF THE DTMS FRAMEWORK

The DTMS Framework consists of a series of base classes that provide the basic behaviors required by the all of the objects in the system, while simultaneously allowing developers to provide the customized behaviors required by their specific application. At the top level, DTMS consists of five base classes, which provide the basis for every other class found in the framework. Each major system component is represented by one of the five base classes: models, solvers, controls, fluids, and the overarching simulation framework. These classes are each discussed in detail in the following sections, and diagrams are provided in Appendix B to illustrate the inheritance relationships between classes of the DTMS Framework. In what follows the words "model" and "component" are interchangeable and synonymous.

3.3 DTMS MODEL CLASS

The topmost base class provided in the DTMS Framework which is used to represent models is known as the `DTMSModel` base class. Most importantly, this class includes the pure virtual functions that provide the basis for the simulation system. These functions contain the physical equations that are used by the model to simulate its physical counterpart, and each is called by the simulation framework during the course of a DTMS simulation.

The `setDefaults` function is the first function that is invoked when any model is created and is responsible for setting the various model-specific variables to default values. Typically, this function is called from the class constructors and should be used to completely initialize the object to a valid, working state including providing default parameter values, allocating any necessary memory, and preparing the model for use with the output system.

The `initialize` function is invoked by the simulation system for every model before the simulation begins, but after the user has provided any customized inputs to the model. Any initial calculations that depend on the input provided by the user should be performed within this function. For example, certain models have geometric calculations that require the physical dimensions of the component and are only performed once. These calculations are placed within the `initialize` function to ensure that any user-specified dimensions are utilized during the initialization of the model before the simulation begins.

The simulation system then calls the `calculateStateDerivatives` and `calculateStates` functions once during each time step of the simulation being executed. Specifically, these functions are used to calculate any time- and system-dependent properties for the model. The `calculateStateDerivatives` function is invoked first and should be used to collect physical properties from any connected models and perform any calculations which do not modify the values of the state variables for the particular model. Typically, this involves acquiring energy domain-dependent physical properties, such as the enthalpy of the upstream model in a thermal fluid system, and calculation of the derivatives of state properties. The `calculateStates` function is invoked following completion of the `calculateStateDerivatives` function and should be used to modify the value of any state variables for the current model. Depending on the physical equations used to represent the physical behavior of a component, equations within this function may involve algebraic calculations to update the properties of the model or integration of the state derivatives calculated in the `calculateStateDerivatives` function.

The `DTMSModel` base class also contains several generic functions that allow data to be communicated between various components within the simulation system. The two

functions provided for this purpose are the `get` and `set` functions. These access methods are used extensively throughout the DTMS Framework for numerous purposes, including in the control system to monitor and modify the values of various state variables and model parameters, in the input system to allow third-party interfaces to properly initialize any DTMS model through a generic and expandable interface, in the output system to define customized output variables, and in the model classes themselves to communicate energy domain-specific data such as the enthalpy between thermal-fluid flow models or torques between inertial shaft models.

Additional functions found in the `DTMSModel` base class are related primarily to the input system, the output system, and the debugging system. Each of these is discussed further in Chapters 5 and 6.

3.3.1 Resistive Network Classes

The `DTMSModel` class is used as the base class for every model created in the DTMS Framework, and therefore it is designed to function independently of any particular modeling method. However, since the Resistive Network modeling strategy is the default modeling platform and provides the basis for all work that currently utilizes the DTMS Framework, the base class `ResistiveNetworkModel` is provided in order to supply models with the functionality needed to adhere to this modeling strategy. This class is derived from the `DTMSModel` base class, and thus inherits all of the methods and behaviors discussed in the previous section.

The primary contribution of the `ResistiveNetworkModel` base class is to provide the means of specifying whether a particular model should be treated either as an independent component of the simulation system or as a dependent component that requires a solver to properly resolve the flow and effort aspects of the model. Any model specified as independent represents a source component of the system and must be able to

calculate and provide the values of its state variables independently from the flow characteristics of the system.

The `ResistiveNetworkModel` base class also contains the complex programming logic that is required to allow Resistive Network models to be connected to one another through the input system and to then be added to the required system solvers. Further details concerning this connection logic are discussed in Chapter 5.

As discussed in Chapter 2, the Resistive Network modeling strategy primarily contains two types of models: flow models which calculate a flow value based on the inlet and outlet efforts, and effort models which calculate an effort value that maintains conservation of flow across its inlet and outlet boundaries. Each of these may function either as dependent upon or independent of the characteristics of the overall flow network. Independent models represent source components and provide the boundary conditions that are necessary to solve an overall flow network. Each type of Resistive Network model has a corresponding base class to represent it in DTMS, which are discussed in the next two sections.

3.3.1.1 Flow Model Class

The primary purpose of the `ResistiveNetworkFlowModel` class is to provide the basic structure that allows model developers to supply a flow equation to represent the physical behavior of their flow model. Since this flow equation is dependent upon the values of the efforts provided by the inlet and outlet models, the model must first have access to the bounding effort models and be able to extract information from them. This access is provided by storing C++ pointers to each of the effort models, each of which must be properly connected by the simulation user before the simulation begins. To aid in this task, two functions are provided in the `ResistiveNetworkFlowModel` class: `setInletEffortModel` and `setOutletEffortModel`. When invoking these functions,

the simulation user provides either a reference or a pointer to the effort model that is connected respectively upstream or downstream of the current flow model. Throughout the DTMS Framework, the terms "inlet" and "outlet" correspond to a nominal flow direction defined by the user and are used interchangeably with the terms "upstream" and "downstream". The distinction between these two terms is insignificant to the solution of the flow network, but will determine whether the flow for a particular model is presented as positive or negative. A positive flow value represents flow that is traveling from the inlet effort model to the outlet model, whereas a negative flow value represents flow traveling from the outlet effort model to the inlet model. Once they have been set by the simulation user, the pointers corresponding to the inlet and outlet effort models can be queried, for various purposes, by invoking the `getInletEffortModel` and `getOutletEffortModel` functions.

Having pointed to the upstream and downstream effort models, the user may now access various data values from these models by using the generic `get` and `set` functions that were briefly discussed in Section 3.3. Since it is integral to the solution of the flow network, the most commonly used data value retrieved from these bounding models will be their respective effort value. Recognizing the frequency with which this value will be retrieved, the `ResistiveNetworkFlowModel` class is designed to automatically retrieve these values at every time step and store them in the scalar `upstreamEffort_` and `downstreamEffort_` variables for follow-on use within the functions of the class. These values may be retrieved or manipulated externally using the `getUpstreamEffort` and `getDownstreamEffort` or `setUpstreamEffort` and `setDownstreamEffort` functions respectively.

Using these effort values, the flow value for the current model is calculated in the `calculateFlow` function. This function is responsible for acquiring any necessary data

from external models, other than the efforts, and for calculating the flow according to the flow equation required to represent that physical model. This flow is iteratively calculated by the network solvers during each time step until the system reaches a quasi-steady-state configuration. Once this has been achieved, the final value for the flow at the current time step is stored in the `flow_` variable and can be accessed and modified externally using the `getFlow` and `setFlow` functions.

In the nonlinear resistive network solvers that are provided with the DTMS Framework, the solution to the flow network is calculated by iteratively solving a matrix equation for the dependent efforts of the system. Each solver uses modified forms of the Newton-Raphson convergence method to quickly and accurately approach the correct solution. However, this method requires additional information about each flow equation in order to properly solve the flow network. Specifically, the first partial derivative of the flow equation with respect to each of the effort variables is required in order to construct the matrix equations that will be used by the system solvers. Thus, the model developer must provide the necessary equations that are used to analytically or numerically evaluate the values of these derivatives in the `calculateFlowPartials` function of the `ResistiveNetworkFlowModel` class. The values for these derivatives are stored in the `upstreamFlowPartial_` and `downstreamFlowPartial_` variables, which are externally accessible through the `getUpstreamFlowPartial` and `getDownstreamFlowPartial` functions and may be externally modified through the `setUpstreamFlowPartial` and `setDownstreamFlowPartial` functions.

Both the `calculateFlow` and `calculateFlowPartials` functions are provided as pure virtual functions in the `ResistiveNetworkFlowModel` base class, and thus these functions must be properly implemented by the specific model classes that are derived from this common base class.

3.3.1.2 Effort Model Class

Effort models in the Resistive Network modeling strategy are based on the principle of conservation of flow across the inlet and outlet models that are connected to them. In order to solve the flow network, effort models must provide the network solvers with the current differential between the total inlet flow and the total outlet flow acquired from the various connected models. Using this information, the solvers can iteratively calculate a more accurate set of dependent effort values until the flow differential at every dependent effort model is zero, at which point the solution of the flow network has been achieved for the current time step.

To calculate the flow differential, the `ResistiveNetworkEffortModel` must have knowledge of the flow models connected to the inlet and outlet boundaries of the model. Unlike the flow models, effort models in the DTMS Framework may have an indefinite number of inlet and outlet connections. In order to reasonably store this information, the `ResistiveNetworkEffortModel` contains two vector data structures which store the pointers for all of either the inlet flow models or outlet flow models that are connected to the current model. These connections must be provided by the simulation user and are assigned using the `addInletFlowModel` and `addOutletFlowModel` functions. The list of flow models that are connected to the current `ResistiveNetworkEffortModel` can be queried by invoking the `getInletFlowModels` and `getOutletFlowModels` functions.

The flow differential for the current `ResistiveNetworkEffortModel` is calculated in the `getTotalFlow` function by requesting the current flow value from each of the connected flow models and then subtracting the total outlet flow from the total inlet flow. Additional functions, `getTotalInletFlow` and `getTotalOutletFlow`, are provided to calculate the total inlet and outlet flows respectively, as their names imply. Once the flow differentials have been calculated, the flow network solvers are

responsible for updating the value of the effort for each dependent effort model contained in the system. This information is then stored in the `effort_` variable inside each `ResistiveNetworkEffortModel` class and can be accessed externally by using the `setEffort` and `getEffort` functions.

The final significant function for the `ResistiveNetworkEffortModel` is the `setEffortsInFlowModels` function, which is called by the network solver once the flow network has been completely solved for the current time step. This function is responsible for updating the `upstreamEffort_` and `downstreamEffort_` variables found in each of the flow models that are connected to the current effort model.

In the vast majority of applications, a model developer will not need to modify the functionality provided by the `ResistiveNetworkEffortModel` for any class that is derived from it. However, all of the functions within this class have been declared virtual in order to permit implementation of any customized behavior required for a specific application. Regardless, in most cases, the physical behavior of an effort-based component can be captured entirely within the `calculateStateDerivatives` and `calculateStates` functions.

3.3.2 Energy Domain Classes

While the resistive network terms “effort” and “flow” are useful for discussing the modeling strategy in a generic sense independent of any particular energy domain, they are not very useful when discussing complex physical systems. For actual application, the generic flow network terms are replaced with their specific energy domain equivalents, such as pressure and flow rate for fluid systems, voltage and current for electrical systems, force and velocity for translational systems, etc. Therefore, the DTMS Framework provides a mechanism for combining the physical properties associated with

various energy domains into the generic flow network architecture provided by the modeling strategy base classes.

These energy domain classes are responsible for collecting the various domain-dependent properties and functionality to properly associate the generic flow and effort models with actual physical components. Each class does not initially derive from any of the existing DTMS base classes, and the properties and methods of these classes may be implemented in whatever manner best suits the specific energy domain. These classes can then be incorporated with the various Resistive Network classes through multiple inheritance, which allows new class definitions to inherit both the properties of the Resistive Network base class and the energy domain class.

Currently, the only energy domain class provided by the DTMS Framework is for use in thermal-fluid flow networks. As this application was the initial primary usage of the DTMS Framework, it was the first to take advantage of the concept of an energy domain class. With the recent introduction of electrical models into the DTMS Framework, the common electrical properties will be collected into an energy domain class in the near future.

3.3.2.1 Thermal Fluid Class

The `ThermalFluidModel` class provided by the DTMS Framework contains a collection of thermal properties that are often related with thermal-fluid flow networks, including properties related to the thermodynamic state and saturation conditions of the fluid, the work and heat input to the model from external sources, and the heat transfer properties between the fluid and the walls of the model. Each `ThermalFluidModel` also contains a pointer to a `DTMSFluid` model, allowing the model to update the internal fluid properties using customized calculation routines based on different types of fluid and various physical assumptions.

The thermodynamic state of a simple compressible substance can be uniquely determined using only two independent, intensive properties. Thus, DTMS employs several functions to update the entire state of the model: `updatePropsPH` allows a model to be updated using the pressure and enthalpy of the fluid, `updatePropsPS` utilizes the pressure and entropy to determine the state, and `updatePropsPT` calculates a non-saturated state from the pressure and temperature of the fluid. Using a single saturated property, the saturated vapor and liquid properties of the fluid can be determined, and thus the `updateSatPropsP` and `updateSatPropsRv` functions are available to calculate these properties based on the current pressure and vapor density, respectively. Once these properties have been determined, the fluid quality can be used to uniquely determine the saturated state of the fluid. Further discussion of the fluid property routines is presented in Section 3.7 and in the thesis of Patrick Hewlett [13].

The `ThermalFluidModel` base class has been combined with several Resistive Network base classes to form the basis for all of the thermal-fluid flow models that have been created for the DTMS Framework. Every effort model designed using the thermal-fluid energy domain is derived from the `ThermalFluidEffortModel` base class, whereas every thermal-fluid flow model is derived from the `ThermalFluidFlowModel` base class. Depending on the structure of the flow equation, certain flow components have been derived from the `ThermalFluidSquareRootFlowModel` base class, which is in turn derived from both the `ThermalFluidFlowModel` base class and the `SquareRootFlowModel` base class. This is discussed in Section 3.3.3.3 below.

3.3.3 Flow Equation Classes

When several flow models are connected in series in a resistive network, additional characteristics for that group of models are often revealed if the components share similar flow equations. In particular, certain types of flow equations can be

combined into a single common equation that can then be used to calculate the flow through the entire series of models without having to solve for the individual efforts in between each flow model. From the perspective of the simulation solver, the ability to reduce the number of dependent effort models within a simulation and simplify the overall system architecture can greatly reduce the amount of computation time required to find the solution to the flow network.

To take advantage of these relationships between flow equations, the DTMS Framework contains several base classes that simplify the process of calculating the flow equation for a model and allow sequences of flow models with common flow equations to be connected in series. These base classes represent only a few of the possible flow equations that can be used in this manner, and future developers are free to add others to fit their desired application.

3.3.3.1 Linear Flow Model Class

In the DTMS Framework, a linear flow model is defined as one which has a flow equation of the following form:

$$f = C \cdot \left((e_i - e_o) + S \right) \tag{3.1}$$

where f represents the flow for the current model, C represents the flow coefficient or conductance, e_i and e_o represent the inlet and outlet efforts of the model, and S represents an effort source term which may correspond to either an internal generation or extraction of effort by external energy sources. Figure 3.10 depicts several flow models with this linear flow equation appearing in series in a notional system:

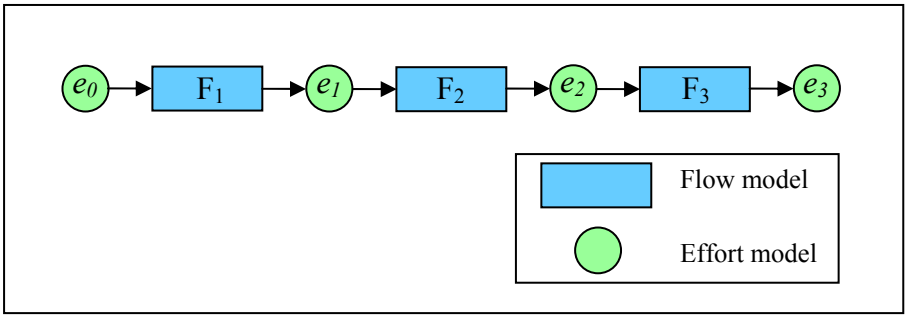


Figure 3.10: Linear flow models in series

Each of the flow models in Figure 3.10 is associated with a linear flow equation as shown:

$$f = C_1 \cdot \left((e_0 - e_1) + S_1\right) \tag{3.2}$$

$$f = C_2 \cdot \left((e_1 - e_2) + S_2\right) \tag{3.3}$$

$$f = C_3 \cdot \left((e_2 - e_3) + S_3\right) \tag{3.4}$$

Due to the conservation of flow throughout the system, the flow passing through each of the flow models is identical. With this knowledge, each equation can be solved for the effort difference:

$$e_0 - e_1 = \frac{f}{C_1} - S_1 \tag{3.5}$$

$$e_1 - e_2 = \frac{f}{C_2} - S_2 \tag{3.6}$$

$$e_2 - e_3 = \frac{f}{C_3} - S_3 \tag{3.7}$$

By then adding each of these equations together, the equations can be solved for the total difference of efforts across the entire series:

$$e_0 - e_1 + e_1 - e_2 + e_2 - e_3 = e_0 - e_3 = \frac{f}{C_1} - S_1 + \frac{f}{C_2} - S_2 + \frac{f}{C_3} - S_3 \tag{3.8}$$

Equation 3.8 can then be solved for the series flow f:

$$f = \frac{1}{\frac{1}{C_1} + \frac{1}{C_2} + \frac{1}{C_3}} \cdot \left(e_0 - e_3 + \left(S_1 + S_2 + S_3\right)\right) \tag{3.9}$$

By substituting a total flow coefficient term C_t for the complex coefficient term in Equation 3.9 and replacing the summation of the individual source terms with a collective total source term S_t, the equation returns to the form of the standard linear flow equation:

$$f = C_t \cdot \left(e_0 - e_n + S_t\right) \tag{3.10}$$

where C_t and S_t represent the general linear series conductance and flow source terms, respectively, presented by the following:

$$C_t = \frac{1}{\sum_{i=1}^{n} \frac{1}{C_i}} \tag{3.11}$$

$$S_t = \sum_{i=1}^{n} S_i \tag{3.12}$$

With knowledge of the flow coefficients and source terms of each of the linear flow models that are connected in series, a single linear flow equation can be generated that eliminates the need to solve for the individual efforts between the flow models. In order to take advantage of this system simplification, the DTMS Framework provides the `LinearFlowModel` base class that is derived from the `ResistiveNetworkFlowModel` base class. This class provides variables to store the flow coefficient and the source term, and it provides calculations for both the linear flow equation and its partial derivatives based on these terms in the `calculateFlow` and `calculateFlowPartials` functions. Linear flow models can be connected in series in a DTMS simulation by creating a linear container model and invoking the `addSeriesFlowModel` function for each of the physical models contained in the series.

3.3.3.2 Nonlinear Flow Model Class

While flow networks consisting entirely of linear flow models may be solved directly using a single matrix equation representing the system, systems of nonlinear components must be solved iteratively using convergence methods. While there are currently no solvers in the DTMS Framework that take advantage of systems of linear flow models, the framework provides the capability to distinguish between models with the two distinct types of flow equations. The `NonlinearFlowModel` base class is used to represent any flow model that has a flow equation that does not conform to the linear flow equation presented in Equation 3.1. This base class provides no additional functionality to any derived classes and is used exclusively to distinguish between linear and nonlinear models.

3.3.3.3 Square Root Flow Model Class

As presented in Section 2.4, one of the most common flow equations for fluid-based systems is represented by generalized form presented in Equation 2.25 and repeated here:

$$f = C \cdot \sqrt{\Delta e + S} \tag{3.13}$$

Using a process similar to the one followed in Section 3.3.3.1, a series of flow models that each contains a flow equation of the square-root form can be simplified to a single square-root flow equation that is capable of calculating the flow through the entire series of models. Equation 3.14 shows the general series flow equation for models with the square-root flow equation:

$$f = C_t \cdot \sqrt{e_0 - e_n + S_t} \tag{3.14}$$

where C_t and S_t represent the general square-root series conductance and flow source terms, respectively, presented by the following:

$$C_t = \sqrt{\frac{1}{\sum_{i=1}^{n} \frac{1}{(C_i)^2}}} \tag{3.15}$$

$$S_t = \sum_{i=1}^{n} S_i \tag{3.16}$$

In the DTMS Framework, the `SquareRootFlowModel` base class is provided to capture this behavior, and it operates in a manner vary similar to the `LinearFlowModel`

presented earlier. The flow equation and flow partials are calculated as functions of the flow coefficient and source terms provided by the user. A series of models with the square-root flow equation can be placed in series by again creating a container model from a `SquareRootFlowModel` object and invoking `addSeriesFlowModel` for each model in the series.

3.4 DTMS SOLVER CLASSES

The system solvers in the DTMS Framework are responsible for calculating the value of any system parameter that cannot be or is not resolved within individual models or components. Generally, these are properties that rely on the overall characteristics of the simulated system, rather than model-specific parameters that can be calculated using locally resident, domain-specific data within the model.

The `DTMSSolver` base class provides the basis for all of the solvers that are used in the DTMS Framework. Aside from basic debugging and input system functionality, the `DTMSSolver` class is responsible for providing the pure virtual functions that allow any solver to be integrated into the overall DTMS simulation architecture. There are three essential functions which must be defined by each solver class that is derived from the `DTMSSolver` base class: `setDefaults`, `initialize`, and `solve`.

Like the `DTMSModel` class, the `setDefaults` function initializes the value of any parameters involved in creation of the solver object. These parameters correspond to default values, memory allocations, and output system parameters that are required by the `DTMSSolver`.

The `initialize` function performs calculations and operations that are required by the solver before the simulation begins execution. This function is called by the simulation system after the user has supplied inputs to the solver object, including adding references to all of the models that must be solved. Typically, this function is used to

prepare the matrices that will be used to calculate the solution of the system by taking advantage of the knowledge of the model topography provided to the solver through the various models in the simulation.

Once per time step, the `solve` function for each solver in a particular system simulation is called upon to calculate system-wide parameters. Often this process involves iterative calculations in order to converge upon a quasi-steady-state solution for the system at the current time step. Classes that derive from the `DTMSSolver` base class may implement customized behavior within this function to resolve the system-dependent variables of a DTMS simulation. A variety of solution methods have been implemented in the DTMS Framework, which are discussed in the following sections. The diversity provided by these solver classes allows the user to select the ideal solution method for the current simulation.

3.4.1 Resistive Network Solvers

In resistive flow networks, each flow model is capable of calculating the value of its flow through the various customized flow equations; however the efforts of the system cannot be calculated individually and must be handled externally by a solver. The `ResistiveNetworkSolver` base class is utilized to obtain references to all of Resistive Network models in the system and then map out the general topography of the flow network.

To accomplish these tasks, the `ResistiveNetworkSolver` class provides a function called `addModel` that stores a pointer for every model in the simulation into various arrays inside the class. Simulations may consist of multiple flow networks, often in different energy domains, and thus it is the responsibility of the user to assign each model to a corresponding solver. Independent flow networks can be managed by different solvers, but every model within a single flow network must be properly assigned

to the single, most appropriate `ResistiveNetworkSolver` object. Internally, the `addModel` function creates separate lists of the dependent and independent flow models and effort models within the entire flow network.

The `initialize` function of the `ResistiveNetworkSolver` class is implemented to acquire the topography of the resistive network and establish all connections between the various models, along with the upstream and downstream relationships between them. This information is vital to proper construction of the matrices that are used to solve the flow networks in the solver classes presented in the next two sections.

3.4.1.1 Newton- Raphson Network Solver

The `NewtonRaphsonResistiveSolver` class provided with the DTMS Framework is a fully-implemented solver that is used to iteratively acquire the solution to resistive networks using the Newton-Raphson convergence method. The specific details of this solution method, including construction of all of the necessary matrices, are implemented in the `solve` method of the class. Since this method requires iterative calculations to converge to the system solution, the user may specify the error tolerance that will be used to determine when convergence has actually been achieved by invoking the `setErrorTolerance` method. Aside from assigning the resistive network models to the `NewtonRaphsonResistiveSolver` object and setting the error tolerance, no further input is required for this solver to be used in a DTMS simulation.

3.4.1.2 Globally-Convergent Network Solver

While the Newton-Raphson method is capable of providing very quick convergence results for nonlinear systems, it requires a sufficiently accurate initial condition in order to achieve peak performance. Without this, the convergence method

may require a considerable number of iterations to achieve the desired solution, or it may never be able to converge to a solution.

In situations where a reasonable initial condition cannot be guaranteed or the system suffers from large variations in operating conditions, the DTMS Framework provides the `GloballyConvergentResistiveSolver` class. This class is derived from the `NewtonRaphsonResistiveSolver` class and performs a series of additional steps during each calculation to ensure that the solution method is converging properly. Specifically designed to accommodate systems with non-ideal initial conditions, this solution method initially performs a Newton-Raphson calculation, and it constantly evaluates the performance of the calculation to ensure that the result is proceeding closer to a desired solution. While this method requires more calculations during each iteration over the standard Newton-Raphson method, in the end it may often significantly reduce the number of iterations that are required in order to produce convergence.

Like the `NewtonRaphsonResistiveSolver` class from which it was derived, the `GloballyConvergentResistiveSolver` only requires information about the models in the flow network and the user-specified error tolerance in order to initiate a system solution.

3.5 DTMS SIMULATION CLASS

The `DTMSSimulation` class is the most important component of any system simulation in DTMS as it is responsible for handling every aspect of a simulation execution including initialization of each component, proper invocation of the simulation functions, and output of the simulation data to a results file. All the timing aspects for a simulation are handled within this class, which are managed through various parameters that must be provided by the user before the simulation begins. The total duration of the simulation is controlled through the `finalTime` variable, which represents the execution

time in simulation seconds. At every time step during the simulation, solvers resolve the flow network and each model invokes the `calculateStateDerivatives` and `calculateStates` functions. The frequency at which this occurs is controlled through the `timeStep_` parameter of the `DTMSSimulation` object, which is measured in simulation seconds. The total number of quasi-equilibrium state calculations that are performed during the simulation can be calculated by dividing the `finalTime_` value by the `timeStep_` value. The final time-based parameter of the `DTMSSimulation` class is the frequency at which the simulation data is output to a file, which is controlled through the `writeStep_` variable and is also measured in simulation seconds. Each of these parameters must be initialized by the user before the simulation begins either by passing these parameters to the `DTMSSimulation` constructor or by invoking the `setFinalTime`, `setTimeStep`, and `setWriteStep` functions.

In order for the `DTMSSimulation` object to properly execute each aspect of the simulated system, this class must have access to every component that exists in the current system, including all of the models, controls, and solvers that are utilized. To access this information, the class provides a specific function for each type of component: `addModel` is used for all classes derived from the `DTMSModel` base class, `addSolver` is used for all `DTMSSolver` classes, and `addControl` is used for any class derived from the `DTMSControl` base class.

All results produced by a DTMS simulation are output to an external data file to allow for further analysis after the simulation has been completed. The `DTMSSimulation` object is responsible for gathering all of the relevant data from the various components in DTMS and properly formatting the data in the output file. To assist in this process, the user must provide the `DTMSSimulation` object with a name for the output file, which can be provided either with the constructor or by invoking the `setOutputFileName` method.

Currently, the data is written in a comma-separated values (CSV) format, which can be easily parsed and analyzed in a variety of graphics-based programs, most notably Microsoft Excel. Further details on the output system are presented in Chapter 6.

3.6 DTMS CONTROL CLASS

While the modeling system of the DTMS Framework promotes the use of self-contained, independent models, the DTMS controls system is composed of a series of building blocks that allow customized controller networks to be created in a manner most applicable to the task at hand. These building blocks can be arranged by the developer or simulation user to create unique feedback control features based on the requirements of an application.

The basis for all control components in the DTMS Framework is the `DTMSControl` class, which provides the basic control object structure to allow the various derived classes to integrate seamlessly with the DTMS simulation system. Aside from various functions associated with input, output, and debugging aspects of the simulation framework, the primary features of the `DTMSControl` class revolve around the calculation of a feedback signal based on a set of controller inputs. This class contains three functions which are responsible for this behavior, each of which is discussed in the following paragraphs.

The `initialize` function of the `DTMSControl` class plays the same role as the `initialize` methods found in the `DTMSModel` and `DTMSSolver` classes in that it allows for the calculation of various internal parameters based on inputs from the user. This function is invoked by the simulation system before a simulation begins execution and is utilized by various control classes to prepare transfer function variables, set initial conditions for time-dependent variables, and establish connections between various internal components.

Calculations that are dependent upon the state of the simulation system are performed each time step in the `simControl` function of the `DTMSControl` object. Whenever this function is invoked, it is responsible for acquiring necessary information from any DTMS components that have been connected to the control and for calculating the output feedback signal based on this information. The behavior captured in this function can be as simple as applying a gain to an input signal or as complex as integrating a series of transfer functions that each vary based on the properties of the system with which they are associated.

Once the output value has been calculated within the `DTMSControl` object, external systems may access this value by invoking the `getOutput` method. This function is primarily used to communicate output signals between various control objects that reside in the same feedback loop and for printing the output value to the simulation data file.

From the `DTMSControl` base class, various control structures have been created to allow developers to construct feedback loops within DTMS simulations for dynamic manipulation of system parameters to account for variations in system behavior. Individual control components for constant and dynamic control sources, adjustable controller gains, model-monitoring tools, feedback summation, and customizable transfer functions are provided with the DTMS Framework. Each of these components may be combined in various ways to create both open and closed feedback loops possessing a wide variety of characteristics required by the system. Control features may also be added as the developer deems appropriate.

3.6.1 DTMS PID Controller Class

The proportional-integral-derivative (PID) controller is one of the most widely-used, generic, control loop feedback constructs for various industrial-level physical

systems. Due to the universal applicability of this type of controller to numerous types of dynamic systems, the DTMS Framework provides a control mechanism to simulate this controller behavior that is built from the control-level building blocks discussed in the previous section.

The structure of a PID controller involves the use of a transfer function to modify a system parameter of a control device based on the error between a metered value in the system and the desired setpoint for that particular variable. The transfer function of a PID controller involves three components: a *proportional* value determines the reaction to the current error, while an *integral* value calculates the reaction based on the accumulation of previous errors, and a *derivative* value reacts to the rate at which the error is changing. A block diagram of the PID controller system is shown in Figure 3.11:

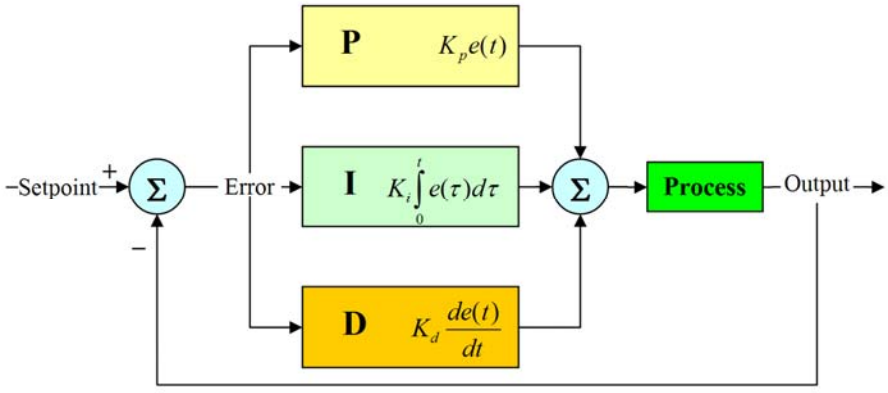


Figure 3.11: Block diagram of a PID controller [21]

In the DTMS Framework, the `CTLPIDController` class captures the behavior of the PID controller by combining each of the required components into a single control mechanism. To utilize this controller in a system simulation, the user must provide a reference to the system variable to be monitored, along with the desired setpoint for this variable, and a reference to the system variable that will be modified by the controller,

including any upper and lower bounds. Once these are in place, each of the P, I, and D constants for the transfer function can be modified to provide the desired system controller behavior.

3.7 DTMS FLUID CLASS

In the early stages of development of the DTMS Framework, all fluid flow models employed constant thermodynamic properties for liquid water only, which greatly limited the applicability and scope of the framework. In order for the fluid flow networks to remain useful for a variety of engineering applications, the capability was added for fluid flow models to easily adapt to custom, user-defined fluid property calculation routines. These property calculators allow the fluid flow components in the DTMS Framework to be utilized with any thermodynamic fluid that is applicable to the current application. Furthermore, the customized fluid routines can utilize a variety of assumptions about the properties of a specific fluid. For example, if a simulation system utilizes incompressible gaseous flow, then property calculators based on ideal gas assumptions can be used in order to speed up the simulation calculations. If incompressible flow cannot be assumed, then the system could utilize fluid property calculators based on empirical data to compute extremely accurate representations of the fluid state. The simulation user is free to select the fluid property calculation routines that provide the best combination of speed and accuracy for the current application.

To implement these fluid property calculation routines, the `DTMSFluid` base class is provided to model developers. This base class contains variables, along with external access and modification functions, for numerous fluid state properties, saturation properties, and chemical properties. Once the current state of a fluid has been calculated, all of these various state and saturation properties within the `DTMSFluid` object are updated to reflect the new thermodynamic conditions. With the chemical properties,

fluid flow models can implement processes that take into account the chemical composition and other chemical features of the current fluid.

Several functions are provided in the `DTMSFluid` class that allow the state of the fluid to be updated by supplying various state variables to the fluid property calculator. These are identical to the functions found in the `ThermalFluidModel` base class that was discussed in Section 3.3.2.1. To update the entire thermodynamic state of the fluid, the following functions can be invoked from the `DTMSFluid` class: `updatePropsPH` allows the model to be updated using the pressure and enthalpy of the fluid, `updatePropsPS` utilizes the pressure and entropy to determine the state, and `updatePropsPT` calculates a non-saturated state from the pressure and temperature of the fluid. Using a single saturated property, the saturated vapor and liquid properties of the fluid can be determined, and thus the `updateSatPropsP` and `updateSatPropsRv` functions are available to calculate these properties based on the current pressure and vapor density, respectively. Once these properties have been determined, the fluid quality can be used to uniquely determine the saturated state of the fluid.

3.7.1 REFPROP Fluid Class

Initial efforts in the development of the fluid property calculation routines in the DTMS Framework involved directly linking the `DTMSFluid` class to the National Institute of Standards Reference Fluid Thermodynamic and Transport Properties (REFPROP) database [18]; however, this approach was deemed to be too slow for general purpose use. Subsequently, table-lookup routines using an interpolation process were selected as the ideal method for returning fluid properties, and this method has been used to implement several fluid property calculators within the DTMS Framework, including those for water, Refrigerant-134A, and air.

The process of creating a fluid property calculator within DTMS involves generating a series of property tables from the NIST REFPROP database for the desired fluid and creating routines to properly interpolate data when it falls between tabulated values. Since each fluid has different ranges of valid states due to differing phase change regions and critical values, each new fluid that is added to DTMS has a unique set of data tables and requires custom routines in order to access those tables. Furthermore, to adequately cover the full range of valid states for a fluid, the DTMS fluid property calculator must contain a significant number of data tables (typically over 50), which must each be generated individually from the REFPROP application and adapted to the syntax of C++. For each of these reasons, the creation of a fluid property calculator in DTMS can be a time-consuming process that must be repeated for each new fluid that is added. The process is described in further detail in the thesis of Patrick Hewlett [13].

Although the above process produces accurate results at very high speed when the DTMS Framework is used in a production environment, it is often excessive and inefficient for use during the developmental stages of a DTMS simulation. Thus, a new interface has been developed for the DTMS Framework that allows the user to directly access the REFPROP fluid property calculators during the preliminary stages of simulation design and later switch to the table-lookup property calculators once the simulation performance becomes an important factor. This interface uses the existing DTMS fluid framework and allows the user complete control over the fluid being used. Fluid routines directly access functionality that is provided by the DLL file associated with the REFPROP application; thus anything that is possible within that application can be accomplished from within the DTMS Framework.

In order to utilize the functionality provided by the REFPROP fluid property routines, a user must simply create an object of the type `REFPROP_Fluid` and initialize it

properly. First, the user must select a fluid that is known to the REFPROP system, which is accomplished by calling the `setFluidFileName` function and passing it the name of one of the fluids located in the fluids directory found in the REFPROP installation directory. All other parameters have suitable default values, but the user may customize these by calling additional functions before requesting property values. The additional parameters directly correspond to those required by the `REFPROP SETUP` subroutine and include the types and mass-percentages of components contained in a mixture, the mixture coefficients, and the reference state for thermodynamic calculations. Furthermore, the user can specify the molar fractions of the various components, the current phase, and the desired root for any individual property calculation that utilizes the REFPROP routines.

The REFPROP fluid property calculators then utilize the existing DTMS fluid interface, which allows them to be used everywhere that the table-lookup property calculators are currently being used. Specifically, this refers to five functions that are used to calculate the fluid properties at a particular state. First, `updateSatPropsP` and `updateSatPropsRv` allow saturated properties to be calculated based on the saturation pressure and the saturated vapor pressure, respectively. These functions access the `SATP` and `SATD` routines in REFPROP that each return the saturation pressure, temperature, liquid density, and vapor density. With these three properties, the remaining fluid properties (enthalpy, entropy, c_v, c_p, and viscosity) are calculated using the common REFPROP temperature-density property routines. The fluid calculators also contain the routines `updatePropsPT`, `updatePropsPH`, and `updatePropsPS`, which allow fluid properties to be calculated at non-saturated conditions using the pressure and temperature, pressure and enthalpy, and pressure and entropy, respectively. These functions utilize the REFPROP routines `TPFLSH`, `PHFLSH`, and `PSFLSH`, respectively, to

calculate most of the fluid properties, and then call `TRNPRP` to calculate the fluid viscosity.

Although the REFPROP fluid property calculators are considerably slower than the table-lookup calculators that have previously been used in the DTMS Framework, these new calculators significantly reduce the amount of time needed to implement a new fluid or a new property routine. Users may customize the desired fluid using the REFPROP calculators without needing to recompile any of the existing code and can switch between different mixture concentrations during the course of a simulation. Furthermore, the error and range checking features provided by the REFPROP system greatly aids the DTMS developer during the simulation design process.

3.8 ADDITIONAL INFORMATION

This chapter provides a general overview of the structure of the DTMS Framework and details many of the base classes that are utilized throughout the modeling system. However, it would be impossible to provide all of the details of the framework within this chapter alone, and thus various other resources are available that present more detailed information.

While the structure and layout of the various classes in the DTMS Framework have been updated throughout its development, many of the technical details of the simulation system, the solver routines, and the controls system have remained the same. In-depth details concerning these aspects of the DTMS Framework are found in [20]. Similarly, development of the fluid property calculation routines and the calculation methods utilized by the various fluids within the DTMS Framework are described in [13]. Finally, additional information concerning the general makeup of a system simulation within the DTMS Framework can be found in Appendix C.

Chapter 4: Inertial and Capacitive Models in DTMS

In generalized flow modeling systems, the characteristics of the flow through the network of models can be classified into three distinct behaviors: resistance effects, inertial effects, and capacitive effects. Resistance effects involve the dissipation of energy through various mechanisms: friction, ambient heat transfer, light, sound, etc, which convert energy from one form to another. Resistance models can be considered as transformer elements that transfer energy across flow networks through various physical processes, often transforming available energy into unavailable forms that exist outside the boundaries of the current system. On the other hand, inertial and capacitive elements act as energy storage devices that collect energy into various often useful forms throughout the flow network. Inertial elements allow energy to be stored through the motion of physical particles, better known as kinetic energy. Capacitive elements allow energy to be stored as potential energy through the displacement of various aspects of the system.

As its name implies, Resistive Network modeling involves the construction of a system of resistance elements to model various behaviors of a physical system. In certain energy domains, such as incompressible fluid flow, large portions of the elements of the system can be modeled solely as resistance elements. During the early stages of the development of the DTMS Framework, these resistance elements accounted for nearly every thermal fluid component that existed in the model library. However, simulations involving various other energy domains, such as electrical and mechanical systems, rely much more heavily upon the energy storage behaviors of inertial and capacitive elements. As the DTMS Framework grew to encompass a broader range of application, the need for

representations of the capacitive and inertial behaviors became an important consideration.

Resistive networks rely on the assumption of steady flow, which disallows the buildup or depletion of energy within components of the system with respect to time. With this restriction, the behavior of inertial and capacitive components cannot be directly simulated using the Resistive Network modeling strategy. Nevertheless, these effects can be modeled *indirectly* by utilizing time-based approximations of the variation in efforts and flows through a particular model to simulate the desired behaviors.

This chapter discusses development of the fundamental equations for inertial and capacitive elements in the DTMS Framework. Included are construction of time-based models that are used to simulate their behavior in the Resistive Network modeling strategy and the representation of various common bond graph structures within the DTMS Framework. As an example, an electrical RLC circuit is constructed to demonstrate the simulated results of these energy storage effects, and a new type of model is discussed that allows for the grouping of the various system behaviors into a single DTMS model representation.

4.1 INERTIAL MODELING IN DTMS

Unlike resistive elements which rely on the relationship between the flow and effort properties of a particular system, inertial elements are largely concerned with the concept of the motion of a system, which is often referred to using the generalized term *momentum*. The concept of momentum is most widely understood in mechanical systems, represented by linear momentum in translational systems and angular momentum in rotational systems. Other types of systems typically represent the momentum based on its relation to the flow variable. Electrical systems refer to

momentum as the infrequently-used magnetic flux linkage, while thermal and chemical systems have no unique variable to represent this quantity.

The generalized momentum is most frequently defined by its relationship to the efforts of a system; specifically, the effort is equivalent to the time-derivative of the momentum, as shown in Equation 4.1:

$$\dot{p} = e \tag{4.1}$$

where p is used to represent the momentum and e represents the effort. Whereas resistance elements are represented by the constitutive relationship relating the flow to the effort, as shown below:

$$f = \phi_R(e) \tag{4.2}$$

inertial elements are represented by a constitutive relationship relating the flow to the momentum:

$$f = \phi_I(p) \tag{4.3}$$

In bond graph theory, the momentum found in Equation 4.3 would become an additional unknown in the system, and the constitutive relationship shown in Equation 4.1 would lead to a set of differential equations that must be solved in order to produce a solution for all variables in the system.

As would be expected, the addition of inertial models into a bond graph network results in a system of unknown quantities that depends entirely upon the makeup of the simulated system. However, the resistive network solvers found in the DTMS Framework have been deliberately designed such that the efforts from the dependent

effort models are the only unknown variables in the system. The elegance and simplicity of this approach are two of the primary reasons that the Resistive Network modeling strategy was chosen for the DTMS Framework, rather than the alternative modeling system offered by bond graph theory.

Nevertheless, combining Equation 4.1 with Equation 4.3, the constitutive relationship for inertial models can be represented as a functional relationship between the flow and the time-integral of the effort:

$$f = \phi_I\left(\int e \cdot dt\right) \tag{4.4}$$

Since this relationship represents the flow as a function of the effort, and thus satisfies Equation 4.2, it can be used within the Resistive Network modeling strategy so long as the time-integral of the effort can be calculated. While this is virtually impossible to accomplish analytically without explicit knowledge of every flow equation in the system, the time-integral of the effort can be calculated numerically using a group of formulas known collectively as the Closed Newton-Cotes formulation [32]. This set of equations consists of a series of numeric integral approximations with increasing orders of accuracy, up to a fourth-order scheme, that are employed based on the amount of information available during a time step. The procedure is described below.

At time step 0, the system knows only the effort for the current time step, e_0, and thus cannot calculate any meaningful integral value. In this case, the inertial model must be provided an initial momentum by the user in order to calculate the flow. During the calculations for time step 1, the inertial model has access to the efforts for both time step 0 and time step 1, and thus it can calculate the approximate momentum using the Trapezoidal Rule:

$$p_1 = p_0 + \frac{h}{2}(e_0 + e_1) \tag{4.5}$$

where p_1 and e_1 represent the momentum and the effort respectively at the current time, p_0 and e_0 represent the momentum and the effort respectively at time 0, and h represents the time step for the simulation. At time step 2, Simpson's Rule is used to calculate the momentum:

$$p_2 = p_0 + \frac{h}{3}(e_0 + 4 \cdot e_1 + e_2) \tag{4.6}$$

where p_2 is the current momentum and e_2 is the current effort. Simpson's 3/8 Rule is used to calculate the momentum at the third time step:

$$p_3 = p_0 + \frac{3h}{8}(e_0 + 3 \cdot e_1 + 3 \cdot e_2 + e_3) \tag{4.7}$$

where p_3 is the current momentum and e_3 is the current effort. Finally, Boole's Rule is used to calculate the momentum at the fourth time step:

$$p_4 = p_0 + \frac{2h}{45}(7 \cdot e_0 + 32 \cdot e_1 + 12 \cdot e_2 + 32 \cdot e_3 + 7 \cdot e_4) \tag{4.8}$$

where p_4 is the current momentum and e_4 is the current effort.

Following the fourth time step, the use of the Newton-Cotes formulas is repeated for all subsequent time steps, where n represents any integer greater than 1:

$$p_{4n+1} = p_{4n} + \frac{h}{2}(e_{4n} + e_{4n+1}) \tag{4.9}$$

$$p_{4n+2} = p_{4n} + \frac{h}{3}\left(e_{4n} + 4 \cdot e_{4n+1} + e_{4n+2}\right) \tag{4.10}$$

$$p_{4n+3} = p_{4n} + \frac{3h}{8}\left(e_{4n} + 3 \cdot e_{4n+1} + 3 \cdot e_{4n+2} + e_{4n+3}\right) \tag{4.11}$$

$$p_{4n+4} = p_{4n} + \frac{2h}{45}\left(7 \cdot e_{4n} + 32 \cdot e_{4n+1} + 12 \cdot e_{4n+2} + 32 \cdot e_{4n+3} + 7 \cdot e_{4n+4}\right) \tag{4.12}$$

Table 4.1 lists the four numerical integration formulas that make up the set of Closed Newton-Cotes formulas, along with the approximation error associated with each. In each formula, the error is a function of the time step; for example, for Boole's Rule, the error is on the order of h^7, where h is the time step. Therefore, the accuracy of these components can be increased by decreasing the size of the time step for the overall system simulation.

Table 4.1: Closed Newton-Cotes Integration Formulas [32]

Numerical Integration Scheme		**Error**
1st-Order (Trapezoidal Rule)	$p_1 = p_0 + \frac{h}{2}\left(e_0 + e_1\right)$	$O(h^3)$
2nd-Order (Simpson's Rule)	$p_2 = p_0 + \frac{h}{3}\left(e_0 + 4 \cdot e_1 + e_2\right)$	$O(h^5)$
3rd-Order (Simpson's 3/8 Rule)	$p_3 = p_0 + \frac{3h}{8}\left(e_0 + 3 \cdot e_1 + 3 \cdot e_2 + e_3\right)$	$O(h^5)$
4th-Order (Boole's Rule)	$p_{4n+4} = p_{4n} + \frac{2h}{45}\left(7 \cdot e_{4n} + 32 \cdot e_{4n+1} + 12 \cdot e_{4n+2} + 32 \cdot e_{4n+3} + 7 \cdot e_{4n+4}\right)$	$O(h^7)$

To ease the process of representing inertial behavior within the DTMS Framework, the `InertialFlowModel` base class has been created to encapsulate nearly all of the required technical details. This class is derived from the `ResistiveNetworkFlowModel` class and provides an interface to the model developer

that mirrors the structure of the flow model class. Whereas the primary functionality for the `ResistiveNetworkFlowModel` class is found in the `calculateFlow` and `calculateFlowPartials` methods that satisfy the constitutive equation for resistive elements found in Equation 4.2, the primary functionality for the `InertialFlowModel` class is found in the `calculateFlowFromMomentum` method and in the `calculateFlowPartialsFromMomentum` method that satisfy Equation 4.3 for inertial elements. Any model that is derived from the `InertialFlowModel` base class must simply provide the proper equations for these two functions in order to capture the inertial behavior of the physical component. Within this class, the upstream and downstream momentum is automatically calculated from the upstream and downstream effort values using the Newton-Cotes formulas, and these are available to the model developer through the `upstreamMomentum_` and `downstreamMomentum_` variables.

To calculate the partial derivative of the flow with respect to effort as required by the nonlinear resistive network solvers, the following relationship is used:

$$\frac{\partial f}{\partial e} = \frac{\partial f}{\partial p} \cdot \frac{\partial p}{\partial e} \tag{4.13}$$

In this equation, the partial derivative of the flow with respect to the effort is separated into the partial derivative of the flow with respect to the momentum and the partial derivative of the momentum with respect to the effort. The partial derivative between the momentum and the effort is calculated directly from the set of Newton-Cotes formulas, whereas the partial derivative between the flow and momentum must be provided by the user through the `calculateFlowPartialsFromMomentum` method.

Although the Newton-Cotes formulas provide a reasonable approximation for the integration required of most inertial elements, the `InertialFlowModel` class has been

designed to allow customized approximations to take the place of default methods that are used. The Newton-Cotes formulas are most accurate in well behaved systems with equally-spaced intervals. Therefore a system that utilizes a variably-sized time step should provide a different approximation routine, such as Gaussian quadrature [32] or Clenshaw-Curtis quadrature [19]. By overriding the functionality found in the `calculateUpstreamMomentum`, `calculateDownstreamMomentum`, and `calculateFlowPartials` methods, a developer can adapt the integration approximations to the requirements of any particular application.

4.2 CAPACITIVE MODELING IN DTMS

While the property inertia is not commonly expressed in the fundamental equations of various types of physical systems, capacitance is more widely used throughout these systems. Representing springs in mechanical systems, capacitors in electrical systems, and storage tanks in fluid systems, the capacitive elements are characterized by the storage of potential energy through the *displacement* of flow.

Similar to the relationship between generalized momentum and effort used to represent inertial behavior, capacitive behavior can be characterized using the following relationship between flow and displacement:

$$\dot{q} = f \tag{4.14}$$

where q represents the generalized displacement and f represents the generalized flow. The constitutive relationship between displacement and effort is defined using the generic capacitance function ϕ_C:

$$q = \phi_C(e) \tag{4.15}$$

Like their inertial counterparts, capacitive elements in bond graph theory introduce additional unknown variables into the network and require solution of differential equations in order to find the overall solution to the simulated network. These additional variables can again be eliminated by combining the results of Equations 4.14 and 4.15 into a single equation that relates the generalized flow to the effort:

$$f = \frac{d}{dt}\left(\phi_C(e)\right) \tag{4.16}$$

This result allows capacitive elements to be simulated using the Resistive Network modeling strategy so long as the time-based derivative can be properly simulated within the capacitive model. Like integral approximations used in the inertial models, it would be difficult to directly differentiate the capacitive function found in Equation 4.16 without intimate knowledge of all flow equations in the network. However, numerical approximations of this derivative can be implemented that allow the capacitive behavior to be simulated without requiring an exact solution to this differential equation.

Although various techniques exist for generating a numerical approximation of a derivative, the nature of the DTMS modeling system prevents the use of data from future time steps, and thus backward difference schemes [31] have been selected for use in the capacitance models of the DTMS Framework. Similar to the inertial element, the capacitive model must be provided with an initial displacement in order to have enough information to calculate the flow at the start of the simulation.

For the first three simulation time steps, the flow through the capacitive element is calculated using first-, second-, and third-order backward differencing schemes, respectively, which are displayed below. At time step 1, the flow is calculated as:

$$f_1 = \frac{q_1 - q_0}{h} \tag{4.17}$$

where f_1 and q_1 represent the flow and displacement at the current time, q_0 represents the displacement at time 0, and h represents the simulation time step. At time step 2, the following equation is used:

$$f_2 = \frac{3 \cdot q_2 - 4 \cdot q_1 + q_0}{2h} \tag{4.18}$$

where f_2 and q_2 represent the flow and displacement at the current time, q_1 represents the displacement at time 1 (the previous time step), q_0 represents the displacement at time 0, and h represents the simulation time step. The calculation of the flow at the third time step utilizes:

$$f_3 = \frac{11 \cdot q_3 - 18 \cdot q_2 + 9 \cdot q_1 - 2 \cdot q_0}{6h} \tag{4.19}$$

where f_3 and q_3 represent the flow and displacement at the current time, q_2 represents the displacement at time 2 (the previous time step), q_1 represents the displacement at time 1, q_0 represents the displacement at time 0, and h represents the simulation time step. After the third time step, the following fourth-order backward differencing formula is used for all subsequent derivative calculations, for any integer n greater than 3:

$$f_n = \frac{25 \cdot q_n - 48 \cdot q_{n-1} + 36 \cdot q_{n-2} - 16 \cdot q_{n-3} + 3 \cdot q_{n-4}}{12h} \tag{4.20}$$

where f_n and q_n represent the flow and displacement at the current time, q_{n-1} represents the displacement at the previous time step, q_{n-2} represents the displacement two time steps in

the past, q_{n-3} represents the displacement three time steps in the past, q_{n-4} represents the displacement four time steps in the past, and h represents the simulation time step.

Table 4.2 lists the four backward differencing equations that are used in the DTMS Framework to simulate the displacement derivative for capacitance-based models along with the approximation error associated with each. In each formula, the error is a function of the time step; for example, for the third-order scheme, the error is on the order of h^3, where h is the time step. Therefore, the accuracy of the capacitive components in the DTMS Framework can be increased by decreasing the size of the time step for the overall system simulation.

Table 4.2: Backward Differencing Equations [31]

Numerical Integration Scheme		Error
1st-Order	$f_1 = \frac{q_1 - q_0}{h}$	$O(h)$
2nd-Order	$f_2 = \frac{3 \cdot q_2 - 4 \cdot q_1 + q_0}{2h}$	$O(h^2)$
3rd-Order	$f_3 = \frac{11 \cdot q_3 - 18 \cdot q_2 + 9 \cdot q_1 - 2 \cdot q_0}{6h}$	$O(h^3)$
4th-Order	$f_n = \frac{25 \cdot q_n - 48 \cdot q_{n-1} + 36 \cdot q_{n-2} - 16 \cdot q_{n-3} + 3 \cdot q_{n-4}}{12h}$	$O(h^4)$

The DTMS Framework implements the behaviors of the capacitance-based element in a base class known as the `CapacitiveFlowModel`. To simulate the capacitive behavior, the developer must create a derived class that supplies the specific capacitance equation, as shown in Equation 4.15. Once this information has been provided in the `calculateDisplacementFromEffort` method, the flow for the capacitive model is calculated by using the backward differencing scheme described.

The nonlinear resistive network solvers again require the values of the partial derivatives of the flow with respect to the upstream and downstream efforts, which is approximated using the following:

$$\frac{\partial f}{\partial e} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial e} \tag{4.21}$$

The calculation of the partial derivative of the flow with respect to the displacement is achieved by differentiating the backward difference equations presented in Table 4.2, which is handled automatically in the `CapacitiveFlowModel` base class. Since the constitutive equation relating the effort to the displacement is provided by the model developer, the partial derivative of the displacement with respect to the effort must also be provided by the user in the `calculateDisplacementPartialsFromEffort` method.

While the backward differencing schemes provided in the `CapacitiveFlowModel` base class are sufficiently accurate for many applications, custom differentiation routines may be provided for use with models in the DTMS Framework. By overriding the functionality of both the `calculateFlowPartials` method and the `calculateFlowFromDisplacement` method, the model developer can adjust the performance of the approximation routines in a class derived from the `CapacitiveFlowModel` base class to best fit the requirements of the desired application.

4.3 DTMS Representations of Common Bond Graph Structures

With the representations of the three types of physical behaviors now separated into distinct base class models in the DTMS Framework, it would not be unexpected to discover a single physical component that encapsulates all three of the behaviors. For example, the primary function of a mechanical spring is to represent the storage of energy through capacitance. However, the coils of the spring also exhibit inertial behavior due

to the mass of the material, and any frictional damping effects within the spring represent resistance behavior. In general purpose modeling systems such as bond graph theory and Resistive Network modeling, these separate physical behaviors are modeled as distinct components even though they correspond to the same physical phenomenon. The freely-structured and flexible nature of the bond graph modeling system makes it easy to present these components in a logical, grouped manner. However, subsequent translation to the Resistive Networking approach is not always immediately obvious due to the simplified nature of the flow and effort models. This section presents a straightforward means of transferring these common multi-behavior components from their bond graph representation to the equivalent Resistive Network representation.

Each of the following examples will focus on grouping of the various component behaviors around the different junction structures found in bond graph systems. First, all three of the physical behaviors may be present surrounding a 1-junction with additional input and output connections as presented in Figure 4.1:

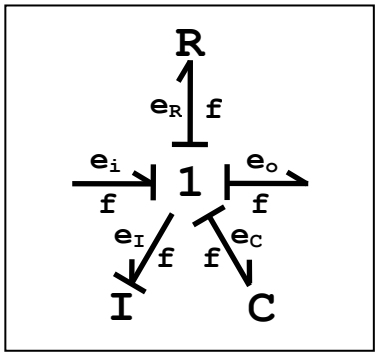


Figure 4.1: Bond graph elements around a 1-junction

As depicted in the figure, the primary characteristic of the 1-junction is to maintain a constant flow throughout all of the branches that connect with the junction.

Each branch also has its own individual effort, with e_i and e_o representing the efforts from an external input and output, respectively.

To represent this structure in a resistive network, the resistance, inertial, and capacitive elements must be converted to the standard bond graph form for a flow model that was presented earlier in Figure 2.4. By placing these elements in series, connected by intermediate effort models, the behavior shown in Figure 4.1 can be achieved:

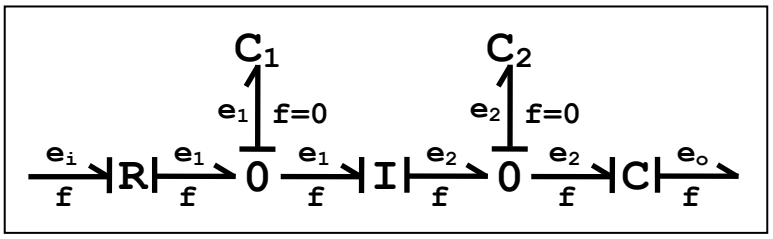


Figure 4.2: Resistive networking equivalent of a 1-junction

Figure 4.2 introduces two new effort values from the intermediate effort models: e_1 and e_2, and these can be related to the e_R, e_I, and e_C effort variables from Figure 4.1 as follows:

$$e_R = e_i - e_1 \tag{4.22}$$

$$e_I = e_1 - e_2 \tag{4.23}$$

$$e_C = e_2 - e_o \tag{4.24}$$

As with the 1-junction, the various physical behaviors can also be arranged around a 0-junction using the additional input and output connections presented below:

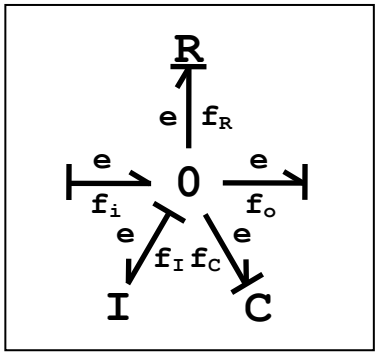


Figure 4.3: Bond graph elements around a 0-junction

Figure 4.3 presents the resistance, inertial, and capacitance elements surrounding a 0-junction, which has the primary attribute of maintaining a constant effort to each connected component while each branch has its own individual flow. In resistive network modeling, this behavior can be achieved by placing the various components in parallel:

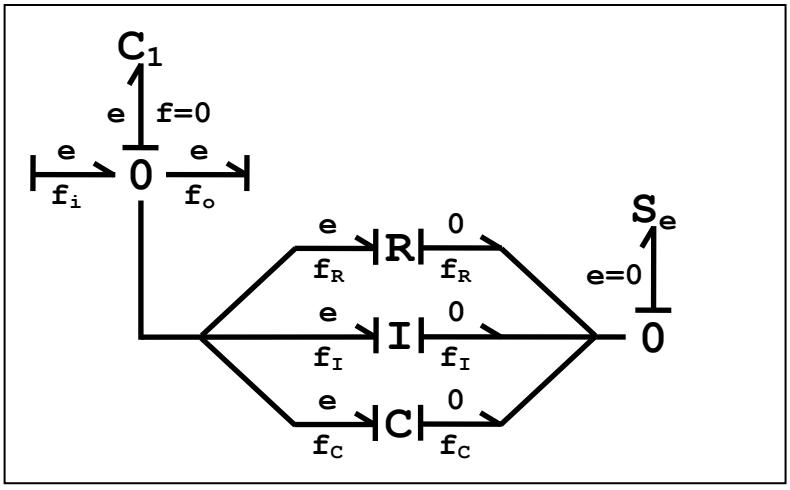


Figure 4.4: Resistive networking equivalent of a 0-junction

Since this structure is much more complicated than that used to represent the 1-junction found in Figure 4.2, it requires a more in-depth explanation. The primary

element associated with this structure is the effort model labeled C_1, which provides the common effort e and contains the connections for the inlet and outlet flows f_i and f_o, respectively. Branching from this effort model are the resistive, inertial, and capacitive flow elements placed in parallel. Since the effort across these elements must be the same as that transmitted to the inlet and outlet flows, each of these flows must in turn be connected to an effort source that acts as a "ground" of zero effort. This ensures that the difference between inlet and outlet efforts across these flow elements is identical to the inlet effort. This configuration requires that the flow equations for each of the resistive, inertial, and capacitive elements be strictly a function of the effort difference across the respective element, rather than any function of the inlet and outlet efforts. However, this assumption is also found implicitly based on the initial configuration around the 0-junction from Figure 4.3 since each element has only a single inlet effort.

4.4 RLC Electrical Circuit Example

Due to the approximate differentiation and integration techniques used in representation of the inertial and capacitive elements in the DTMS Framework, it is appropriate to examine the performance of these models against a known physical system. For this demonstration, an RLC electrical circuit is simulated using the DTMS Framework, and the results are compared against the known analytical solution for the system. Although this system consists solely of simple, linear components, more thorough verification of these approximation techniques is demonstrated using complex systems of nonlinear components presented in the thesis of Matthew Pruske [23].

The typical RLC circuit consists of a resistor, inductor, and capacitor placed in series around a closed circuit that is driven by a voltage source. Figure 4.5 presents an electrical schematic for this type of circuit:

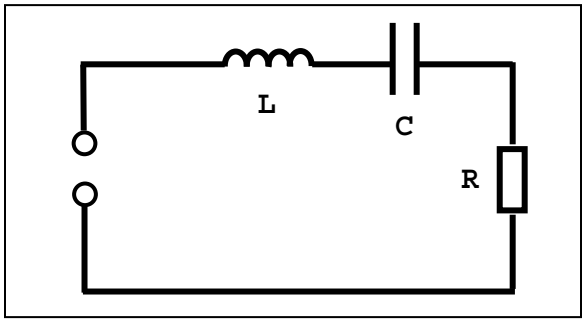


Figure 4.5: Electrical diagram for RLC series circuit

The particular order of the resistor, capacitor, and inductor are unimportant in this representation; they may be rearranged differently within the circuit without affecting the solution, so long as they remain in series with the other system elements. Each component is assumed to be linear and they are represented using the following equations for the resistor, capacitor, and inductor respectively:

$$V = i \cdot R \tag{4.25}$$

$$V = q \cdot \frac{1}{C} \tag{4.26}$$

$$\lambda = i \cdot L \tag{4.27}$$

where for the circuit V represents the voltage across the resistor and represents the effort, i is the electrical current and represents the flow, q is the charge and represents the displacement, λ is the magnetic flux linkage and represents the momentum, R is the linear resistance, C is the linear capacitance, and L is the linear inductance of the electrical network.

The solution for this system is found by summing the voltage drops around the various components of the circuit. Although the voltage drop across the inductor is not immediately apparent from Equation 4.27, it can be easily determined by differentiating

both sides of the equation with respect to time and recognizing that the time-derivative of the momentum is the voltage. The equation for the summation of voltage drops around the entire circuit is shown below:

$$L \cdot i' + R \cdot i + \frac{1}{C} \cdot q = 0 \tag{4.28}$$

where i' is the time derivative of the current. By recognizing that the current is the time-derivative of the charge for the system, the entire equation can be converted into a second-order differential equation with respect to the charge:

$$L \cdot q'' + R \cdot q' + \frac{1}{C} \cdot q = 0 \tag{4.29}$$

Using standard solution techniques for differential equations, the solution to Equation 4.29 is:

$$q = A \cdot e^{m_1 \cdot t} + B \cdot e^{m_2 \cdot t} \tag{4.30}$$

where m_1 and m_2 are determined by substituting back into Equation 4.29 and solving the resulting quadratic equation to produce the following:

$$m_1 = \frac{-R + \sqrt{R^2 - \frac{4 \cdot L}{C}}}{2 \cdot L} \tag{4.31}$$

$$m_2 = \frac{-R - \sqrt{R^2 - \frac{4 \cdot L}{C}}}{2 \cdot L} \tag{4.32}$$

The constants A and B are determined from the initial conditions used to parameterize the problem. For this sample demonstration, the following values will be used:

Table 4.3: Input parameters for RLC circuit

RLC Parameters for equations 4.30, 4.31, and 4.32	
R	3 ohms
C	2 farads
L	4 henries
q (t=0)	500 coulombs
q′ (t=0)	0 amperes

Using these parameters, the values of m_1 and m_2 are calculated as -0.25 s^{-1} and -0.5 s^{-1} respectively, and the values of A and B are determined to be 1000 coulombs and 500 coulombs, respectively. The resulting equations for q and q' when including these calculations are as follows:

$$q = 1000 \cdot e^{-0.25 \cdot t} - 500 \cdot e^{-0.5 \cdot t} \tag{4.33}$$

$$q' = -250 \cdot e^{-0.25 \cdot t} + 250 \cdot e^{-0.5 \cdot t} \tag{4.34}$$

With the analytical solutions complete, the results of the companion DTMS simulation can be evaluated to determine the overall accuracy of the approximations used for the inertial and capacitive behaviors in the DTMS Framework. Figure 4.6 presents the results of both the analytical solution and the DTMS approximate solution using a time step of 0.1 seconds for both the electrical charge q and the system current i or q'. In this figure, the numerical solution from the DTMS simulationlies directly on top of the analytical solution and thus cannot be distinguished from it graphically. Figure 4.7 plots the absolute error in the DTMS approximations compared with the analytical solution throughout the course of the simulation.

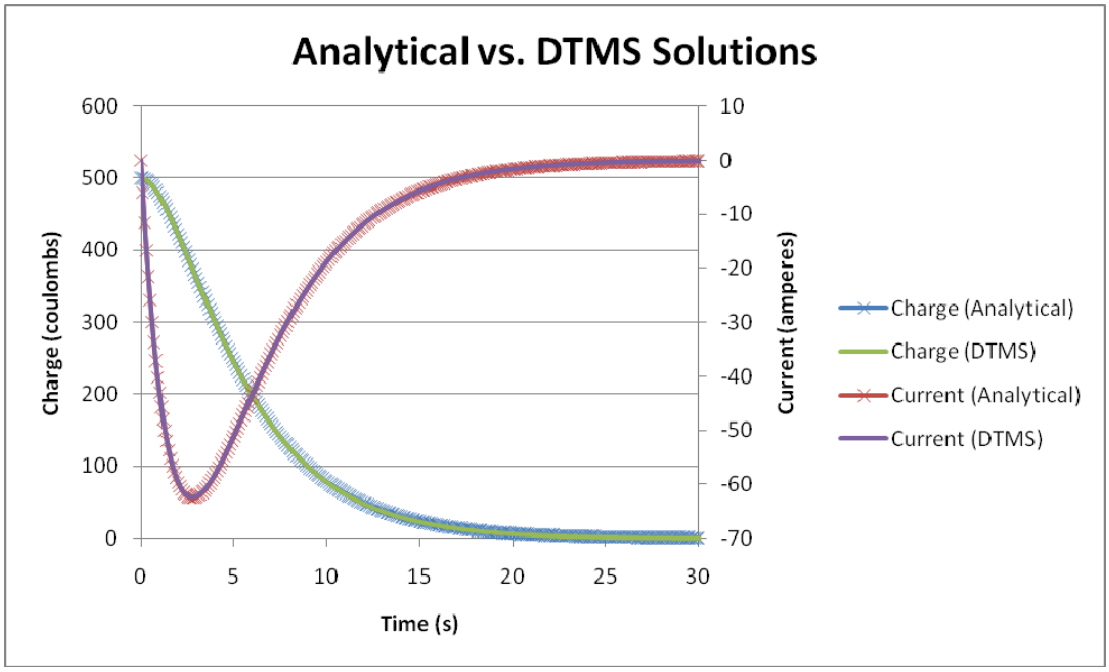


Figure 4.6: Analytical vs. DTMS Solutions to RLC Circuit

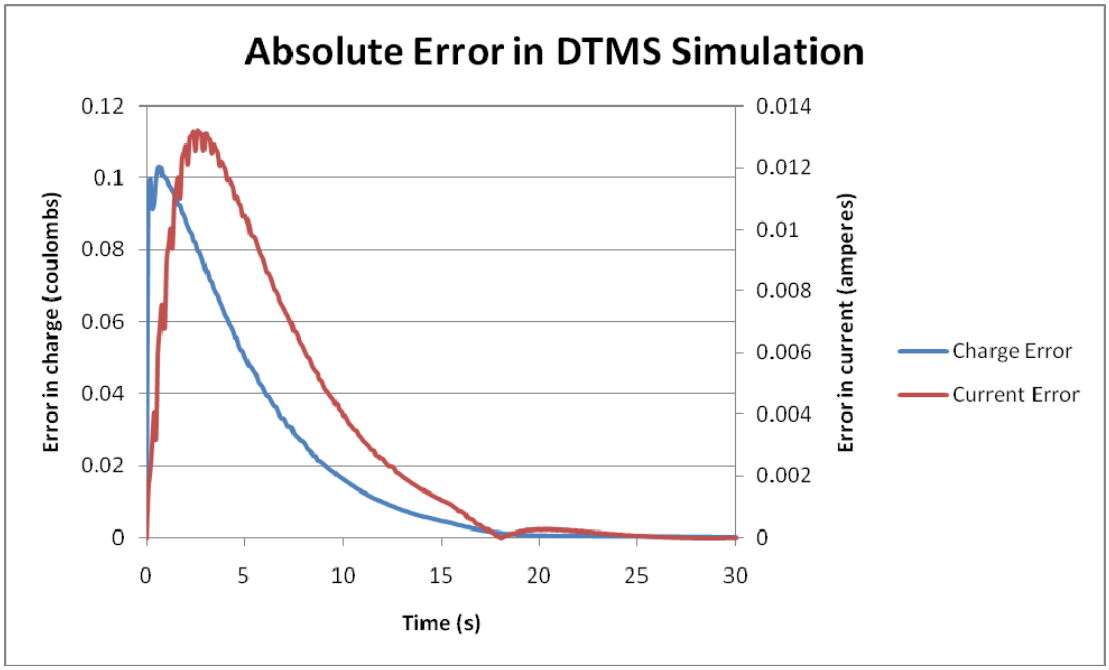


Figure 4.7: Error in the DTMS simulation of the RLC Circuit

Figure 4.7 demonstrates a very small absolute error between the analytical result and the numerical result using the DTMS Framework. The error in the electrical charge from the capacitor peaks just above 0.1 coulombs, which represents slightly more than 0.02% relative error. Similarly, the error for the current through the system peaks slightly above 0.013 amperes, which also represents approximately 0.02% relative error. The overall error of the DTMS simulation is controlled by an error tolerance supplied by the user via the system solver. As the current and charge approach zero, the relative error increases dramatically when these values drop below the supplied error tolerance; however, the absolute error continues to decrease as the system approaches steady state. All of these values are well within engineering tolerances and demonstrate, for this example, the relative accuracy provided by the approximations of the capacitive and inertial behaviors within the DTMS Framework.

4.5 CONTAINER MODEL IN DTMS

In some situations, a single physical component may embody multiple resistive, inertial, and/or capacitive behaviors, and it would be easier to present this collection of behaviors to a simulation user as a single, grouped model, rather than the separate DTMS components that are used to represent it. As mentioned earlier, a mechanical spring consists primarily of capacitive effects, but the coils of the spring begin to exhibit inertial behavior as the mass increases, and frictional resistance effects are always present in such a system. Additionally, complex physical machines often consist of multiple individual components that each performs a wide variety of different actions. During the development and testing of this type of model, it is useful to treat it as a collection of individual models; however, once the model has been thoroughly tested and properly validated, it becomes more effective to treat the collection of models as a single unit which can easily be inserted into a larger system model. Thus, it is apparent that there is

a need for sections of a simulation network to be grouped together and treated as a single unit in order to ease the user's interaction with the framework.

Models of this nature differ from the series models which were addressed with the `LinearFlowModel` and `SquareRootFlowModel` base classes in Chapter 3. While those models are intended to reduce the complexity of a system by combining specific flow models to use a single flow equation, these new grouping models merely capture and encapsulate a specific section of a preexisting network model.

To accomplish this purpose, a new class was created within the DTMS Framework called the `ResistiveNetworkGroupModel`. This class contains separate arrays for the flow models, effort models, and group models that are contained within the encapsulated model. While flow models and effort models are only allowed to connect to other models in specific ways (flow models can only have a single inlet effort model and a single outlet effort model, and effort models can only connect to inlet flow models and outlet flow models), group models may be connected in any way that the model developer determines to be useful. Once these external connections have been made, the group model will function exactly as it would if all of the individual models had been manually created, parameterized, and connected in the simulation main file.

The process for creating a new group model from a collection of individual models has been designed for minimal effort on the part of the model creator. Once the individual models have been properly parameterized and configured, the model developer may simply convert the working simulation main file into a group model by copying the parameterization code for the individual models into the `setDefaults` function of the group model. The only additional requirement is to define the proper connection functions that will allow the group model to interact with external models.

The group model further allows the model developer to customize which parameters may be accessed by external users and which variables will be printed to the output file. However, default behaviors have been defined for each of these operations, and thus the user is not required to define any additional functionality in order to make completely functional group models. To demonstrate the usage of the `ResisitiveNetworkGroupModel` class, Appendix E contains the source code for the construction of a group model that encapsulates the behavior of a Rolls-Royce MT30 gas turbine engine.

Chapter 5: Universal Input System for the DTMS Framework

The DTMS Framework was originally designed to function independently of any interface, and it was intended that the code not be altered in any substantial way for a specific software application. Previous development strictly adhered to this philosophy as all simulation users interacted directly with the existing DTMS functions in order to create and execute various simulations. The only method for creating a DTMS simulation was to generate a standard C++ main file with the necessary initialization and execution commands, and then compile and link this using the existing DTMS code base.

During the early stages in development of the DTMS Framework, the primary focus of the project centered on production of meaningful simulations and realistic results. While the interfaces of the framework were recognized as crucial aspects of the future of DTMS, the development of these interfaces was not given a high priority until operational aspects of the framework had been put in place. As the DTMS Framework has grown and matured, the importance of external integration has become more and more important to its continued use and development.

In order to address this growing need for interfaces that integrate with external systems, a simple, expandable, and scalable input mechanism has been developed for direct C++ interaction with the DTMS Framework. This interface is not influenced by any specific external system, but allows other software applications to more easily generate and execute DTMS simulations without requiring that the external system make significant changes as DTMS evolves.

5.1 Improvement of Data Communication System between Models

In its first iteration, the `DTMSModel` class of the DTMS Framework featured a pair of functions, `get` and `set`, which allowed for the communication of various data values

from the model to other parts of the simulation system, including other models and the control system. To allow these functions to be universally applicable across all models in the DTMS model library and remain useful to any additional models which are created in the future, they were carefully designed to allow the user of the functions to explicitly specify which data variable should be accessed or modified within the model.

Initially, the `get` and `set` functions utilized a unique C++ construct called an *enumeration* to uniquely identify every possible data value. C++ enumerations allow the developer to create a set of named integer constants that are evaluated at compile time. These constants can be grouped and assigned a type name, which allows variables to be created that only accept the enumeration constants as valid arguments. Since they are evaluated during the compilation of the program instead of during execution, they can be copied and utilized very quickly in comparison to other data types. The following provides an example of how the developer might create and use an enumeration within a program:

```
enum demo_data
{
      MY_VALUE_1,
      MY_VALUE_2,
      MY_VALUE_3
};
```

Figure 5.1: Sample C++ enumeration

The above code demonstrates how an enumeration is declared. The `enum` keyword tells the C++ compiler that an enumeration is being defined, and `demo_data` is given as the name of the enumeration. This name is then subsequently used to define variables containing copies of the enumeration constants. Finally, the values

MY_VALUE_1, MY_VALUE_2, and MY_VALUE_3 represent the three constants that are part of the demo_data enumeration; each is assigned a unique integer value that represents it.

To use an enumeration, the developer simply creates a variable using the name of the enumeration as the type name, and then assigns it one of the values defined in the enumeration definition:

```
demo_data myVariable = MY_VALUE_1;

if (myVariable == MY_VALUE_2)
   myVariable = MY_VALUE_3;
```

Figure 5.2: Example declaration and usage of a C++ enumeration

In the above example, a variable named myVariable has been declared with the data type demo_data to indicate that it can only accept the constants that were defined in the enumeration definition from Figure 5.1; myVariable is then assigned the value MY_VALUE_1. The only valid values that can be placed in myVariable are MY_VALUE_1, MY_VALUE_2, and MY_VALUE_3. As shown in the next line of Figure 5.2, the value stored in myVariable can then be compared with the enumeration values (in this case, MY_VALUE_2) and assigned a new enumeration value. Most often, enumerations are used in functions with switch statements, as shown in Figure 5.3.

In the initial design of the DTMS Framework, a global enumeration structure named data_t maintained a list of every possible data value that could be utilized by the get and set functions. These functions each accepted an enumeration variable as their first argument to specify which data value was to be accessed or modified. Each model in the DTMS Framework was then responsible for setting up the proper switch structure to relate the enumeration variables with the variables contained in the model. This

technique is depicted in Figure 5.4 which contains the `get` function for the `ResistiveNetworkFlowModel` class.

```
void demoFunction(demo_data myVariable)
{
   switch(myVariable)
   {
   case MY_VALUE_1:
      /* do first action */

   case MY_VALUE_2:
      /* do second action */

   case MY_VALUE_3:
      /* do third action */
   }
}
```

Figure 5.3: Function utilizing a C++ enumeration

```
double ResistiveNetworkFlowModel::get(data_t variable)
{
      switch(variable)
      {
      case FLOW:
            return flow_;

      case UPSTREAM_FLOW_PARTIAL:
            return upstreamFlowPartial_;

      case DOWNSTREAM_FLOW_PARTIAL:
            return downstreamFlowPartial_;

      case UPSTREAM_EFFORT:
            return upstreamEffort_;

      case DOWNSTREAM_EFFORT:
            return downstreamEffort_;

      default:
            return ResistiveNetworkModel::get(variable);
      }
}
```

Figure 5.4: C++ enumeration used in the `get` function

The `ResistiveNetworkFlowModel` class contains five data variables which can be accessed through the `get` and `set` functions: flow, upstream and downstream flow partials, and upstream and downstream efforts. As shown in Figure 5.4, each of these variables has a corresponding enumeration value indicated with all capital letters, and by specifying the proper value as the function argument variable, the desired data value is returned by the `get` method. If an enumeration value is given that does not correspond with one of the known values for the `ResistiveNetworkFlowModel` class, then the default case is applied, in which the enumeration value is simply passed on to the get method of the base class `ResistiveNetworkModel`.

The methods utilized by the `get` and `set` functions allow the user to access any number of internal data variables through a single interface and add both flexibility and customization features to the DTMS Framework. However, the nature of the C++ enumeration obviates the usefulness of this feature as a universally applicable approach in a growing, multi-user, development environment. Specifically, the `data_t` enumeration structure must be completely known to each model in the DTMS Framework during the compilation process. As new models are added to the framework and additional data variables are added to the `get` and `set` functions, the `data_t` enumeration must be updated to reflect these changes and, as a result, every model in the DTMS Framework must also be recompiled. While this merely presents an inconvenience in an open source code environment, it is an impossible task in a closed source environment, where models are compiled individually by their creators and the underlying source code is not available to be modified or recompiled.

Since limitations of the C++ enumeration greatly decrease the usability of the DTMS Framework, a new data structure called the `DTMSData` class has been developed to provide the same functionality as the enumeration structure while promoting independent

development and closed-source design principles [17]. Functionally the `DTMSData` class operates using the same principles as the C++ enumeration. Each object is assigned a unique integer, which allows the object to be copied, utilized, and compared quickly and easily. However, these integer identifiers are assigned at runtime, rather than at compile time, thus allowing greater freedom in creation of `DTMSData` objects while maintaining performance rivaling that of the C++ enumeration. Figure 5.5 demonstrates how the `get` function of the `ResistiveNetworkFlowModel` class utilizes the `DTMSData` object.

```
double ResistiveNetworkFlowModel::get(const DTMSData & variable)
{
      if (variable == FLOW)
            return flow_;

      else if (variable == UPSTREAM_FLOW_PARTIAL)
            return upstreamFlowPartial_;

      else if (variable == DOWNSTREAM_FLOW_PARTIAL)
            return downstreamFlowPartial_;

      else if (variable == UPSTREAM_EFFORT)
            return upstreamEffort_;

      else if (variable == DOWNSTREAM_EFFORT)
            return downstreamEffort_;

      else
            return ResistiveNetworkModel::get(variable);
}
```

Figure 5.5: `DTMSData` class used in the `get` function

Unlike the universal list maintained for the enumeration structure, each `DTMSData` value exists as a separate C++ object that can be created wherever it is most useful, thus eliminating the need for developers of new models to have access to the source code of other parts of the DTMS Framework. To ensure that each `DTMSData` object contains a unique integer identifier, all `DTMSData` objects contain a pointer to a global list that contains references to every `DTMSData` object that has been created. This list is created

automatically and maintained by the system at runtime, thus requiring no additional effort for the model developer.

When a new model is created using the DTMS Framework and new data variables must be created for use with the `get` and `set` functions, additional `DTMSData` objects may be created within the same source file as that used for implementation of the model. To prevent accidental modification and promote compiler optimizations, these new objects are declared as constant structures by adding the C++ keyword `const` before the declaration of the object. Figure 5.6 shows how a `DTMSData` object is declared:

```
const DTMSData UPSTREAM_EFFORT("upstreamEffort");
```

Figure 5.6: Declaration of a `DTMSData` object

Each `DTMSData` object also has a unique name associated with it, which is provided by the user in passing a string to the constructor as shown in Figure 5.6. Unlike the C++ enumerations, the global list of `DTMSData` objects can be searched based on this unique name parameter and can return a reference to the `DTMSData` object that corresponds to the given name parameter. This functionality is used heavily as part of the DTMS input system that is presented later in this chapter.

While providing the same basic functionality as the C++ enumeration, the `DTMSData` class allows greater flexibility in the development of the DTMS Framework and provides additional functionality that was previously unavailable. By allowing each `DTMSData` object to be declared and initialized independently, developers now may supplement and extend the functionality of the `get` and `set` functions for their own model classes without requiring access to the code for other parts of the DTMS Framework. With the ability to search and query the global list of `DTMSData` objects, advanced

initialization routines can now be created that are automatically expanded to any future additions to the DTMS model library without requiring additional effort on the part of the model developer.

The power and versatility of the `DTMSData` class has been employed frequently throughout the DTMS Framework and is explored in more depth in the remaining sections of this chapter and in the output system presented in Chapter 6.

5.2 C++ Class Factories for the DTMS Framework

One of the most important features that allows the DTMS Framework to integrate with external software platforms is dynamic creation of elements of the DTMS Framework at runtime. As originally designed, the execution of a DTMS simulation required the creation of a C++ main file containing the initialization of the DTMS components, model parameterizations and connections, and simulation invocation. This main file was compiled along with the rest of the code from the DTMS Framework, and thus the compiled simulation was permanently fixed with respect to the contained models, the parameterization values, and the component connections. By transferring the steps required to build the DTMS simulation from the compilation process to the program execution process, external software systems are able to dynamically create and execute DTMS simulations without performing the tedious steps to recompile the system.

Within an object-oriented programming environment, the process of creating objects at run time is accomplished by creating a special class that can selectively generate dynamically-allocated class structures based on a customizable user-supplied input. These special classes are commonly referred to as *object factories* [15], and they operate based on a class hierarchy and the principles of polymorphism. In particular, factory classes consist of a creation function that is capable of producing any class object derived from a common base class. The user must supply an input variable to the

creation function that identifies which derived class object is to be created. Since the creation function is only capable of producing base class pointers, polymorphic behavior must be properly implemented in each of the derived classes to allow them to be completely initialized using the functions provided in the base class structure.

To allow dynamic creation of the components of the DTMS Framework, factory classes have been created for each of the five primary data types that are discussed in Chapter 3: `DTMSModel`, `DTMSControl`, `DTMSSolver`, `DTMSFluid`, and `DTMSSimulation`. Each of these classes implements a polymorphic function called `loadState` that allows any classes derived from these base classes to be properly initialized with any necessary data. This function is discussed further in Section 5.3.

However, one of the most important aspects of the DTMS Framework is the ability to connect models together and apply system solvers based on the modeling strategy that is used to simulate the physical system. The association of models with the proper solvers and the rules for valid connections are unique to each modeling strategy and cannot be enforced at the base class level provided by the `DTMSModel` and `DTMSSolver` classes. For this purpose, separate factories must also exist for each of the base classes that are provided by a particular modeling strategy. For Resistive Network modeling, object factories have been created for the `ResistiveNetworkFlowModel` class, the `ResistiveNetworkEffortModel` class, the `ResistiveNetworkGroupModel` class, and the `ResistiveNetworkSolver` class.

In general, factory classes rely on maintaining a list of possible derived classes that are capable of being created. Any time a new class is added to the class hierarchy, the corresponding factory must be updated in order to allow objects of the new class type to be created through the factory interface. In the DTMS Framework, the maintenance of this list presents a problem for individual model developers. The typical approach for

object factories involves manually updating a permanent list of possible derived classes within the factory class object. However, in the DTMS Framework code system, the model developer should not need to have, and in many cases will not have, direct access to various portions of the framework, including the interfaces for the factory classes.

To overcome access control limitations of the DTMS Framework, the lists of possible derived class types for each of the factory classes must be created at runtime and provide a simple, easy method for new derived classes to be added that does not require modifying the factory classes directly. A solution for this lies in the construction of a *pluggable* architecture for the object factory interfaces that are used in the DTMS Framework. In software-based systems, pluggable designs involve the use of *plugins* which exist as separate objects that provide a portion of customized functionality to a larger system [1]. The overarching system is responsible for collecting and utilizing the functionality of the plugins, but the actual programming logic resides entirely within the plugins themselves. For a pluggable object factory, each plugin is responsible for the creation of a single derived class type, while the object factory itself is responsible for collecting the various plugins and executing the proper plugin code based on input supplied by the user. By separating out the class creation code to individual plugins, the code for the object factories remains fixed while individual developers of new DTMS components can implement their own plugins that are added to the factories at runtime.

For each of the object factories in the DTMS Framework, the proper derived class that is to be created by the factory is indicated by passing a string containing the name of the class to the creation function of the factory. Therefore, each plugin associated with the factory must have a unique string name that identifies the class that is created by that particular plugin. When the user provides the string name of a particular derived class, the object factory is responsible for locating the plugin that corresponds to the desired

class, invoking the object creation code from the plugin, and returning a base class pointer to the newly-created derived-class object.

In order to properly integrate with the various factories that have been implemented into the DTMS Framework, any newly-created DTMS component class must include a string variable that represents the name of the class and a plugin for the proper DTMS object factory. Figure 5.7 demonstrates how these variables are incorporated into a typical DTMS class:

```
class Pipe : public ThermalFluidSquareRootFlowModel
{
private:
	static const std::string className_;

public:
	static const ResistiveFlowModelFactoryPlugin<Pipe> plugin;

	//...
};
```

Figure 5.7: Usage of a DTMS factory plugin

The private specification for the `Pipe` class contains a string variable that is used to identify this particular class in several aspects of the DTMS Framework, most notably the object factory system. The factory plugin is declared as a public variable using the most appropriate plugin type. In this case, the `Pipe` class is derived from the `ResistiveNetworkFlowModel` base class, and thus it contains a plugin of the type `ResistiveFlowModelFactoryPlugin`. Since any DTMS model class derived from the `ResistiveNetworkFlowModel` class must also be derived from the `DTMSModel` base class, the `ResistiveFlowModelFactory` automatically adds its plugins to the `DTMSModelFactory` as well, and thus the `Pipe` class does not need to also include a `DTMSModelFactoryPlugin`. Each of these variables is declared as both `static`, which

ensures that a single copy of the variable is shared among all instances of the class, and `const`, which ensures that the value of the variable will not be changed during the course of the simulation and thus allows the compiler to perform certain optimizations concerning the behavior of this variable. The plugin variable utilizes a C++ programming technique referred to as *templates*, which allow a particular data type to be passed to a function or class definition. In this case, the derived class type for the plugin is passed during the creation of the plugin by placing `Pipe` in the angle-brackets after the plugin type.

In addition to the declaration statements found in Figure 5.7 for the class name string and the factory plugin, static class variables in C++ must also provide a concrete initialization within the implementation file for the class. Figure 5.8 demonstrates how the variables in Figure 5.7 would be initialized in an implementation file for the `Pipe` class:

```
const string Pipe::className_ = "Pipe";
const ResistiveFlowModelFactoryPlugin<Pipe>
                                    Pipe::plugin(Pipe::className_);
```

Figure 5.8: Usage of a DTMS factory plugin

The `className_` variable must be initialized with the string that represents the name of the class, which is simply "`Pipe`" in this case. The `plugin` variable is then initialized using this class name by passing it to the constructor of the plugin class.

The factory plugins that have been implemented in the DTMS Framework are designed such that they will automatically register themselves with the appropriate factories whenever a particular plugin is created. Therefore, model developers only need to implement one of the factory plugins in their own class definitions, as shown in Figures 5.7 and 5.8, in order to take advantage of the object factory system that is used

heavily by the DTMS input system. However, the nature of the independent factory plugin system does not require that the plugins be created along with the definition of the derived class. If a developer encounters a DTMS component that has not been properly integrated with the existing factory system or if additional factory classes are developed for the DTMS Framework, additional plugins may be developed outside of the existing class definitions that allow any DTMS derived component to be created through a conforming factory class.

5.3 UNIVERSAL DTMS DATA STRUCTURES

While the object factory system designed for the DTMS Framework is fully capable of handling the dynamic creation of the DTMS components at runtime, it cannot handle the initialization of those objects in the same generalized manner. To accomplish this task, a separate set of data structures has been created to facilitate the transfer of initialization data into the components of the DTMS Framework.

The `DTMSIOObject` base class forms the basis for all of the various classes that are used as part of the input system to transfer data from an external system into the individual DTMS components. Each `DTMSIOObject` has two primary data members:

- `ID`: a unique numerical identifier that is used to distinguish this object from other DTMS objects. This is primarily useful for establishing connections between models, controls, and other DTMS objects.
- `type`: a string variable which holds the class name of the DTMS object that is represented by the current `DTMSIOObject`.

The final data member contained within the `DTMSIOObject` class is a Standard Template Library (STL) map that uses STL strings for both the key and the value. STL maps function similar to C++ arrays, but they allow the data structure to be indexed by any sortable variable type instead of integers. The value that is used as the index into the map

is referred to as the *key*, while the value stored at a particular index is referred to as the *value*.

For the `DTMSIOObject`, this map is used to contain all of the other data that will be associated with a DTMS object. A string holding the unique name of the data variable is used as the key for the map, and the value of the data variable is stored as the corresponding map value. The individual DTMS objects are responsible for assessing the validity of the data held in the `DTMSIOObject` map.

In addition to the `DTMSIOObject`, subclasses have been created to represent each of the five primary object types that are found in the DTMS Framework: `DTMSIOModel`, `DTMSIOControl`, `DTMSIOFluid`, `DTMSIOSolver`, and `DTMSIOSimulation`. These classes do not add any additional data beyond what is provided by the `DTMSIOObject` base class and are designed merely to allow differentiation between the various types of objects within the framework.

To interact with these new classes, the existing base classes and certain derived classes in the DTMS Framework must provide a `loadState` virtual function, which takes a `DTMSIOObject` subclass of the corresponding type as an argument. Each class is then responsible for initializing itself from the object that was passed to it. An example of a typical implementation of the `loadState` function is presented in Figure 5.9:

```
class ExampleModel : public DTMSModel
{
private:

	bool sampleBool;
	int sampleInt;
	double sampleDouble;
	string sampleString;

public:

	virtual void loadState(const DTMSIOModel & inputObject,
		DTMSFactory & factory)
	{
		//Calling the base class method
		DTMSModel::loadState(inputObject, factory);

		//Iterator that will be used to find data variables
		DTMSIOModel::DataMap::const_iterator currentData;

		//Checking for required parameter sampleBool
		currentData = inputObject.data.find("sampleBool");
		if(currentData == inputObject.data.end())
		    throw Parameterization_Error("sampleBool not found");
		sampleBool = convertStringTo<bool>(currentData->second);

		//Checking for required parameter sampleInt
		currentData = inputObject.data.find("sampleInt");
		if(currentData == inputObject.data.end())
		    throw Parameterization_Error("sampleInt not found");
		sampleInt = convertStringTo<int>(currentData->second);

		//Checking for optional parameter sampleDouble
		currentData = inputObject.data.find("sampleDouble");
		if(currentData == inputObject.data.end())
		{
		    sampleDouble =
			    convertStringTo<double>(currentData->second);
		}

		//Checking for optional parameter sampleString
		currentData = inputObject.data.find("sampleString");
		if(currentData == inputObject.data.end())
		{
		    sampleString = currentData->second;
		}
	}
};
```

Figure 5.9: Example implementation of the `loadState` function

The example in Figure 5.9 demonstrates several aspects of this input mechanism. Any class derived from `DTMSModel` will have a `loadState` function which takes a `DTMSIOModel` object as an argument. Note that this function is declared virtual so that any class derived from `DTMSModel` can be properly initialized from this function, even through a base class pointer. The `DTMSFactory` object that appears as the second argument to the function contains a list of pointers to previously-created DTMS objects and their unique IDs, and it can be used to connect the current model with other DTMS objects.

The first action the `loadState` function performs is to call the `loadState` function of its most immediate base class. This ensures that all of the variables for the model are properly initialized from the `DTMSIOModel` class, including any base class variables to which the derived classes may or may not have direct access.

`DTMSIOObject` subclasses may hold both required and optional data elements. However, these classes do not need to enforce or even recognize the difference between these two types as each specific DTMS object is responsible for enforcing its own required data members. This is demonstrated in the `ExampleModel` class in Figure 5.9 which requires the `sampleBool` and `sampleInt` variables and may optionally use the `sampleDouble` and `sampleString` variables. The `loadState` function can use the built-in methods of the STL map class to search the `DTMSIOModel` for the data members that it requires, and if those required elements are not found, an error can be reported to the user. In the above example, a C++ exception class is thrown with a message describing which required data member was not found. The exception classes that have been created to indicate error conditions in the DTMS Framework are discussed further in Chapter 6.

Finally, each data value stored in the `DTMSIOObject` data map is stored using a string-based representation, rather than the native C++ data type of integer, floating-point number, etc. In order to convert these into their native data types, two global conversion functions have been added to aid DTMS developers with this process: `convertFromTo` and `convertStringTo`. The `convertFromTo` function takes two template arguments and uses string streams to convert the function argument from the first template type to the second template type. The `convertStringTo` function is a specialization of the `convertFromTo` function in which the function argument is always a string. These functions can be used for any native C++ data type and for any class type which overloads the stream extraction and insertion operators.

The `loadState` function implemented in the `DTMSModel` base class is unique when compared to the other component base classes in the DTMS Framework. In the `DTMSModel` base class, this function analyzes each of the variables found in the data map inside the `DTMSIOModel` and determines if any of the names of the data variables match the names of any of the `DTMSData` variables that have been created. If a match is found, then the `loadState` function of the `DTMSModel` base class calls the appropriate `set` method using the supplied data value. Therefore, for most models derived from the `DTMSModel` base class, the model developer does not need to provide a specialized implementation for the `loadState` function and merely needs to set up the appropriate `DTMSData` variables and associate them properly in the `get` and `set` functions of the model class.

The generic nature of the `DTMSIOObject` and its subclasses allow these to be used for all existing DTMS objects and for future DTMS objects. By avoiding placement of any model-specific details within the `DTMSIOObject`, both internal DTMS developers and external developers wishing to integrate with DTMS have a common interface

mechanism which will be kept consistent throughout development cycles to allow rapid integration and product updates as both partners continue their growth and expansion.

5.4 DTMS INPUT SYSTEM

The input system designed for the DTMS Framework is fully capable of creating, initializing, and connecting all of the various components of a DTMS simulation using the programmatic interface provided by the systems presented in this chapter.

To execute a DTMS simulation, an external program first creates the necessary components by calling the creation function from the appropriate factories and providing them with the string name representing the derived class object that is to be created. The necessary data to initialize each object is then transferred into corresponding instances of the subclasses of the `DTMSIOObject` base class. Once these objects have been created, the `loadState` function for each of the DTMS components is called by passing it the appropriate `DTMSIOObject` class that contains the initialization data. For any class derived from the `DTMSModel` base class, the data from the `DTMSIOModel` object is transferred into the model using the appropriate `set` functions with the `DTMSData` objects that correspond to each of the data values. At this point, the DTMS system has been completely initialized and is ready for execution to produce the simulation results. Chapter 7 demonstrates how this input system is utilized to integrate the DTMS Framework with the FireGUI graphical simulation software developed by Mississippi State University.

Chapter 6: Output System and Development Tools

In addition to the ability to dynamically configure, initialize, and execute complex physical simulations through a flexible and sophisticated input system, one of the most important features of any simulation software framework is the ability to extract constructive, meaningful results from the outputs produced by the various aspects of the simulation system. Output data includes both analytical results that are used to form meaningful conclusions about the physical behavior of the simulated system and the flow of data through the simulation architecture. In addition, of deep interest is how customized DTMS component classes respond to the various interacting aspects of the simulation framework which are in turn crucial for the construction and verification of accurate physical models, fluids, and control systems in the DTMS Framework.

Every simulated system produces a unique set of outputs that are of interest to the current application. Simulation users need the ability to narrow the wide selection of physical data produced by a DTMS simulation in order to extract the most relevant information from the various physical behaviors that are modeled in the system. Furthermore, the results of DTMS simulations must be prepared in a manner suitable for use in external plotting tools to aid in the analysis of the data and improve the quality of the conclusions that are drawn from these results.

During the development of new, customized components for the DTMS Framework, developers require a set of tools to aid in the process of translating the simulated behavior of various system components from appropriate DTMS class structures. Despite lack of direct access to various internal components of the DTMS Framework, developers must still be able to follow the flow of data throughout different aspects of the simulation system in order to analyze and verify the proper behavior of

their individual, customized additions to the DTMS Framework. Developers also require a means of tracking down errors and unexpected behaviors that become an inevitable part of any development process.

Discussed in this chapter are several additions to the DTMS Framework that are designed to directly address the requirements of both the simulation user and the DTMS developer to extract meaningful data from the results of a DTMS simulation. Section 6.1 discusses a customizable output system that simulation users may utilize to filter and organize the data produced during a DTMS simulation. Developers may also take advantage of the standardized debugging system discussed in Section 6.2 to improve the performance of customized models, controls, and fluids during the development of new components. Finally, as discussed in Section 6.3, users, developers, and external software tools all benefit from the addition of C++ exception classes to explicitly locate and describe any errors which may occur during the execution of a DTMS Simulation.

6.1 IMPROVEMENT OF THE DTMS OUTPUT SYSTEM

The DTMS Framework was initially designed with a simplistic output system that allowed model developers to specify the exact data and format to be produced when a simulation generated output data. While this approach was a convenient and necessary starting point for the framework, a more sophisticated solution quickly became necessary as the user-base expanded and DTMS was targeted for integration with other projects.

As originally designed, each model created for the DTMS Framework contained two functions: `writeHeader` and `writeData`. These functions were designed to allow models to write developer-specified data to a data file in a standardized format. The primary format chosen was the comma-separated values (CSV) format, in which the data is separated by commas to indicate different columns and separate lines to indicate different rows. At the beginning of the simulation, the `writeHeader` function is

responsible for writing the column headers to the output file, while `writeData` is responsible for writing the corresponding model data at the end of each time step.

This original output system was designed to allow some future expansion in terms of customized output formats by allowing the user to change various write flags within each model. Initially, a flag value of 0 would indicate that a particular model should not output any data, while a flag value of 1 would indicate that data should be written in the CSV format. Future developers could define different values for the flag that would indicate new formats; for example, a flag value of 2 could indicate an XML format that could be used to integrate DTMS with an external data visualization program.

However, this approach has several shortcomings which made it necessary to expand and improve upon it. First, in order to add a new output format to the DTMS Framework, the developer was required to modify the `writeHeader` and `writeData` functions of every existing model to output their data in the new format. At best, this task would be tedious, time-consuming, and potentially error-prone in a completely open source code environment, but it would be completely impossible in a closed source environment, since the developer does not have access to the source code for many of the existing models. Furthermore, it becomes the responsibility of the simulation user to ensure that all of the models have the same output format specified. If several models output their data in one format while others utilize a second format, any person or program trying to parse the output file would find it difficult, if not impossible, to decipher the file. The simulation user also has no control over what data values were output by each model as this decision was left solely in the hands of the model developer. Finally, model developers have the responsibility of ensuring that their models output the data in the correct format, resulting in many different sources for potential error.

This chapter discusses enhancements to the DTMS output system that focus on eliminating each of the above shortcomings to enhance the overall output experience for both simulation users and model developers. Rather than require the model developer to manually output the desired data variables in the proper format within the `writeHeader` and `writeData` functions, developers and users are now able to simply specify which variables should be output and how these variables should be labeled. The process of printing the data in the proper format has been moved into the simulation executive, and thus model developers are no longer required to concern themselves with any particular data format nor must they directly interact with the output process in any way.

The new output system is centered around a new data type called `DTMSWriteVariable` which consists of three primary data members:

- A pointer to the current model to which the remaining members apply,
- A `DTMSData` variable that indicates which property of the current model should be output,
- A string which represents the label that is used to identify the property in the output file.

In general, it is not necessary for either the model developer or the simulation user to ever interact with the `DTMSWriteVariable` data type directly. Every model in the DTMS Framework now has a member function called `addWriteVariable`, which takes over the responsibility of creating and maintaining the `DTMSWriteVariable` objects within the model.

The model developer uses the `addWriteVariable` member function to specify properties that will be output by default if the user chooses to make no further changes to the output list. Using this mechanism, the output produced will mimic the behavior of the previous output system, where the model developer had sole control over which

properties are output to the data file. Most commonly, the model developer indicates the default output properties of a model by calling the `addWriteVariable` function within the `setDefaults` function as shown in Figure 6.1:

```
void ThermalFluidEffortModel::setDefaults()
{
      addWriteVariable(PRESSURE,"P");
      addWriteVariable(ENTHALPY,"h");
      addWriteVariable(TEMPERATURE,"T");
}
```

Figure 6.1: Example use of `addWriteVariable` within the `setDefaults` function

In the new output system, the simulation user now has the power to specify which properties are to be printed to the output file. After creating an instance of model, the simulation user may also call the `addWriteVariable` function in order to include additional variables in the output file along with the default data. Alternatively, the simulation user may completely clear the default developer-defined output list by calling the `clearWriteVariableList` function and then specify properties to be output for the current model with calls to the `addWriteVariable` function. Figure 6.2 demonstrates this functionality:

```
ThermalFluidEffortModel exampleEffortModel;
exampleEffortModel.clearWriteVariableList();
exampleEffortModel.addWriteVariable(DENSITY,"R");
exampleEffortModel.addWriteVariable(ENTROPY,"s");
exampleEffortModel.addWriteVariable(EFFORT,"e");
```

Figure 6.2: Example use of `addWriteVariable` by the simulation user

Each model maintains a list of `DTMSWriteVariable` objects that are used to specify the desired output for that particular model. While the simulation is running, the simulation executive has the responsibility of obtaining the desired data from the models

and sending that data to the output file in the required format. The data from the models is obtained by calling the generic `get` function of the model and passing it the `DTMSData` variable that was defined in the `DTMSWriteVariable`. Therefore, the model developer must ensure that the `get` function has been properly defined so as to return all of the necessary data for the model; however, this is the originally designed behavior of the generic `get` and `set` functions, and thus is not a new requirement for the model developer.

The simulation executive is now the only object responsible for ensuring that the output file is created using the proper format, which greatly decreases the likelihood of errors being made and increases the ease of maintenance of the output system. Presently, the output specifications for the CVS file format are hard-coded into the simulation executive, making it the only format used for the output file of a DTMS simulation. Future work will create a class-based system for various output formats that allows new file formats to be added with ease and that also allows the simulation user to specify the desired format for the current simulation.

6.2 STANDARDIZED DEBUGGING SYSTEM

Up to this point, all debugging statements used in the DTMS Framework have been included on an ad-hoc and as-needed basis with no consistency as to what information would be included or how this information would be formatted. Debugging statements were added and removed as various developers required access to different sets of information during their own respective design processes. The work presented in this section describes a standardization of the debugging process to make it simpler for developers to both add to and interpret debugging information produced during a DTMS simulation.

The debugging process should be limited to the development and validation cycle; and end product code should be free of any overhead introduced by the debugging system. Thus, the first and necessary element of any debugging system should be the ability to quickly and easily eliminate debugging code from the compiled executable. Therefore, a generic debug capability has been introduced into DTMS. In the `DTMSDefinitions.h` file, included in every file in the DTMS Framework, the developer now has the option of defining the preprocessor macro `INCLUDE_DEBUG_STATEMENTS`. If this macro is defined, then all of the prepositioned debugging code will be included in the resulting compiled executable. If left undefined, then the debugging code will be automatically stripped from every file during the compilation process and will produce no additional overhead in the resulting executable. To ensure that future code also conforms to these requirements, all debugging code must now be wrapped inside the following preprocessor statements:

```
#ifdef INCLUDE_DEBUG_STATEMENTS
   /* debugging statements */
#endif
```

Figure 6.3: Preprocessor statements to selectively include debugging statements

To aid developers in tracking down when and where most bugs appear, a standard set of functions have been added to the DTMS Framework for the purposes of debugging. The global object `DEBUG` has been created to handle these debugging processes. By default, the debugging system will print all debugging statements to the output stream defined by `std::clog`, which is found in the standard `iostream` header file. The user may assign a different output stream to the `DEBUG` object before a simulation begins by calling the `setOutputStream` function as shown in the following example:

```
ofstream logFile;
logFile.open("debuggingExample.log");
DEBUG.setOutputStream(logFile);
```

Figure 6.4: Sending debugging statements to a log file

However, since the most common use of this function will be to send the debugging information to another file, a separate function has been created which only requires the name of the logging file:

```
DEBUG.setOutputFile("debuggingExample.log");
```

Figure 6.5: Alternative method for sending debugging statements to a log file

One of the most common debugging tasks is to monitor the function calls that execute while a program is running, and the debugging system in DTMS has several functions that are specifically designed to facilitate this task. First, there is the `DEBUG.entering` function which provides the class name, object name, function name, and function arguments to the debugging system. The first parameter, `debuggingLevel_`, will be discussed later:

```
DEBUG.entering(debuggingLevel_, "className", "modelName",
      "functionName", "functionArguments");
```

Figure 6.6: Usage of the `DEBUG.entering` function

To simplify the use of this function, each class in the DTMS Framework contains a private string variable associated with it called `className_` to hold the name of the class, as discussed in Section 5.2. This function call will produce the following line of output to the debugging log file:

```
[className] modelName: <functionName> Entering
functionName(functionArguments)
```

Figure 6.7: Logging statements produced by the `DEBUG.entering` function

There is also a corresponding function called `DEBUG.exiting`, which will produce a similar line of output except with the term `Entering` replaced with `Exiting`. This is extremely useful for developers to see exactly when embedded functions both begin and end throughout the course of a simulation. Whenever `DEBUG.entering` is called at the beginning of a function, `DEBUG.exiting` must be called at the end of the function in order to ensure that the debugging system maintains the proper function information.

The next two functions, `DEBUG.input` and `DEBUG.output`, allow the developer to print the contents of various input and output parameters. The `DEBUG.input` function allows the developer to ensure that a function is receiving the proper information before it begins its operations, while the `DEBUG.output` function allows the developer to verify that the function has performed its operations properly. These functions are demonstrated in the example shown in Figure 6.8.

This example produces output to the debugging log file that appears similar to the statements found in Figure 6.9. In this example, the `doSomething` function is called using the values 4, 439.74, and '`g`' for `variable1`, `variable2`, and `variable3` respectively, and the calculated value of `variable4` is 12.6.

```
void doSomething(int variable1, double variable2, char variable3)
{
#ifdef INCLUDE_DEBUG_STATEMENTS
   DEBUG.entering(localDebugLevel_, className_, modelName_,
      "doSomething", "int, double, char");

   DEBUG.input("variable1", variable1);
   DEBUG.input("variable2", variable2);
   DEBUG.input("variable3", variable3);
#endif

   double variable4;
   /* perform some calculations */

#ifdef INCLUDE_DEBUG_STATEMENTS
   DEBUG.output("variable4", variable4);

   DEBUG.exiting();
#endif
}
```

Figure 6.8: Example using the debugging functions

```
[className] modelName: <doSomething> Entering doSomething(int,
double, char)
[className] modelName: <doSomething> Inputs:
[className] modelName: <doSomething>   variable1 = 4
[className] modelName: <doSomething>   variable2 = 439.74
[className] modelName: <doSomething>   variable3 = g
[className] modelName: <doSomething> Outputs:
[className] modelName: <doSomething>   variable4 = 12.6
[className] modelName: <doSomething> Exiting doSomething(int, double,
char)
```

Figure 6.9: Debugging statements printed to log file

Lastly, the debugging system provides a function for printing additional output within the body of a function. This allows the developer to print out specific statements that might be useful during the development process, but are not represented in the input and output parameters of the function. The usage of the `DEBUG.print` statement is shown in the following example:

```
DEBUG.print("Beginning calculations at time " +
      DEBUG.toString(currentTime));
DEBUG.print("Current error = " + DEBUG.toString(error));
```

Figure 6.10: Example use of the `DEBUG.print` function

As seen in the example above, the `DEBUG.print` statement takes a string as an argument and simply prints it to the debugging log file. However, the developer may want to include additional data in the statement, such as the value of a particular variable. To accommodate this, the `DEBUG.toString` function can be used to convert its argument to a string and concatenate this value with other strings to form the argument for the `DEBUG.print` function. This function utilizes C++ templates and string streams to convert its argument to a string, and therefore can accommodate any data type or class that overloads the << operator.

While having access to all of this information can be very useful during the development process, the quantity of information produced in the debugging log file can quickly become unwieldy during execution of a DTMS simulation. Therefore, the user has been given greater control over how much information is printed and which models may print data. There are four levels of output from which the user may choose:

- `NO_OUTPUT`: No information will be sent to the debugging log file.
- `PRINT_STATEMENTS_ONLY`: Only the information found in `DEBUG.print` statements will be sent to the log file.
- `FUNCTION_CALLS`: The log file will display any information from `DEBUG.print` statements and from `DEBUG.entering` and `DEBUG.exiting` functions. The `DEBUG.input` and `DEBUG.output` functions will be ignored.
- `FUNCTION_CALLS_WITH_IO`: All debugging information will be sent to the log file.

The debugging level can be set in two ways. First, there is a global debugging level which can be set by calling the `DEBUG.setGlobalDebugLevel` function and passing it the desired level as a parameter. For example, the following code would be used to include all debugging information in the log file:

```
DEBUG.setGlobalDebugLevel(FUNCTION_CALLS_WITH_IO);
```

Figure 6.11: Example use of the `DEBUG.setGlobalDebugLevel` function

The second method allows the user to set the debugging level in each individual model in the DTMS simulation. After creating a model, the user may call the `setDebugLevel` method for the new model and pass the model-specific debugging level as an argument. In order to prevent a particular model from printing debugging information to the log file, the following code would be used:

```
exampleModel.setDebugLevel(NO_OUTPUT);
```

Figure 6.12: Example use of the `setDebugLevel` function for DTMS components

In the `DEBUG.entering` function, the first parameter is used to pass the debugging level of the current model to the debugging system so that the proper output will be produced for the current model. By default, all models will use the global debugging level unless they are specifically passed a new level through the `setDebugLevel` function.

The freedom afforded by a user-definable, model-specific debugging system that does not require recompilation adds significant overhead to a DTMS simulation, even if no output is being produced. Therefore, developers who do not wish to use the

debugging system may recompile the DTMS Framework without the debugging statements by simply commenting out the `#define INCLUDE_DEBUG_STATEMENTS` line in the `DTMSDefinitions.h` file.

6.3 ERROR HANDLING SYSTEM

In many programming languages that are not object-oriented, the common method for indicating errors or failures from inside a function is to return an error code, which is typically represented by an application-specific integer value. Developers are often forced to manually check these error codes to ensure that specific function operations succeeded, manually clean up allocated resources if failure did occur, and then examine lengthy error tables to determine the cause of the problem. While a C++ developer may continue to use the error-code paradigm in the design of their applications, a more graceful object-oriented method is available to aid both the developer and the user of an application in the form of C++ exceptions.

Exceptions allow the developer to halt program execution whenever an exceptional condition occurs during the normal operation of the application. However, unlike error codes, exceptions do not need to be checked manually, and they can hold important contextual information about the problem that occurred. Furthermore, well-designed programs can be constructed in such a way that resource de-allocation is performed automatically in the presence of exceptional conditions, without requiring any additional effort on the part of future developers who use and expand upon the existing code base [27].

The DTMS Framework has been constructed to take advantage of the exception mechanism available in C++, and a set of DTMS-specific exception classes have been created to aid developers in the design and debugging of future DTMS expansions. All of the DTMS exception classes have been derived from the standard C++ base class

`std::exception`, which allows users and developers to capture and handle DTMS exceptions alongside any standard system exceptions that may be thrown during the execution of a program. A common base class called `DTMS_Exception` has also been created to allow users and developers to limit the handling mechanisms to strictly DTMS-specific issues. This class holds a string-based message, which allows developers to provide context-specific information about the exception conditions that were encountered. Five primary subclasses have been created to address specific issues that may occur during the execution of a DTMS simulation. Each of these subclasses has been used throughout the DTMS Framework and may be implemented by other DTMS developers into their own objects. These subclasses are described in the following paragraphs.

The first type of exception is named `File_Error` and is designed to be used when the program encounters an error in the handling of a file during the course of a DTMS simulation. This class can optionally hold the name of the file that caused the exception and can display the name to the user when requested. This exception is currently thrown whenever the system fails to properly open a DTMS-required file, which can occur when the `DTMSSimulation` object tries to open the simulation output file and when the debugging system attempts to open the debugging output file. Most commonly, this occurs when the output file has been opened by another application, and thus the DTMS system can ask the user to close the file before retrying the simulation. Figure 6.13 demonstrates how the `File_Error` exception class is utilized in the `initialize` function of the `DTMSSimulation` class:

```
void DTMSSimulation::openFile()
{
      outputFile_.open(fileName_);
      if(!outputFile.is_open())
            throw File_Error("Unable to open file", fileName_);
}
```

Figure 6.13: Use of the `File_Error` exception class in the `DTMSSimulation` class

The `Argument_Error` exception class can be used when the argument to a function does not match preconditions that are required. Normally, this type of exception is most useful when detected during the debugging stages of the program design and thus should be handled by C++ assertions rather than exceptions. However, since the DTMS Framework has been designed to integrate with external applications which may be providing the user input mechanisms, this exception class allows the external application to catch and handle any argument errors that may occur. Like the `File_Error` exception class, the `Argument_Error` class can optionally store the value of the argument which failed to meet the preconditions and display that value to the user. The general-purpose input mechanism described in Chapter 5 uses this type of error message to ensure that the connections between models and controls are valid within the current modeling system.

Similar to the `Argument_Error`, the `Parameterization_Error` can be used to ensure that a user has properly initialized all of the required model parameters before a simulation begins. This is currently used by many models and controls in their `loadState` functions to indicate when the `DTMSIOObject` subclass used to initialize the model does not contain all of the required parameters. It is also used to ensure that specific models and controls contain any required connections, e.g., `ThermalFluidModel` objects must have a `DTMSFluid` associated with them, and `TransferFunction` objects must be connected to a `DTMSControl` object.

The `Fluid_State_Error` is used to indicate when the thermodynamic variables used to specify the state of a particular fluid are invalid for any reason, which could occur under many circumstances including if critical values for the fluid are exceeded or the specified state is outside the valid range for the approximation methods being used. The invalid state property values can be stored within this exception class so that they may be displayed to the user and corrected if necessary.

The final primary DTMS exception class called `Simulation_Error` is used to indicate any general system errors that could occur during the course of a simulation. Presently, this exception is thrown when any of the output variables reach a value of infinity or an undefined value, and when one of the nonlinear solvers fails to show convergence during the simulation.

These new exception classes are the primary means for signaling that an error has occurred during a DTMS simulation. Current and future DTMS developers and DTMS integration partners should make use of this mechanism as a clean and easy way to indicate, capture, and recover from any exceptional situations that might occur within the DTMS Framework.

Chapter 7: Integration with FireGUI Graphical User Interface

While initially used solely by the thermal management team at the University of Texas for complex system simulations, interest in the DTMS Framework has grown throughout the ESRDC as consortium members recognize both the usefulness of the simulation system and the variety and quality of the results being produced. The need to expand and improve upon the DTMS Framework to allow for external interaction and greater ease-of-use for developers and users has been recognized, and the work presented in this thesis has achieved great progress toward meeting this need.

In collaboration with Mississippi State University, the first steps have been taken toward integration of the DTMS Framework with an external graphical modeling tool to greatly expand the user-experience for individuals seeking to utilize the power of the DTMS Framework. The improvements and enhancements to the framework presented in this thesis have provided the backbone for this collaboration and demonstrate the powerful integration capabilities designed into the DTMS Framework.

This chapter begins by describing details of the FireGUI graphical user interface as it was originally developed to visually represent simulations of fire and smoke propagation onboard naval ships in Section 7.1. The details of the sequence of technologies that have been developed to allow integration between the DTMS Framework and the FireGUI interface are presented in detail in Section 7.2, and finally, Section 7.3 presents the results of the integration efforts through the simulation of a simplified mixing-tank cooling loop.

7.1 Initial Design of the FireGUI Graphical User Interface

Fire hazards represent one of the most dangerous threats aboard any modern naval vessel. As a result of highly efficient design practices utilized in the construction of U.S.

Navy surface ships and submarines, closely-packed systems represent a severe hazard if compartmentalized fires are left untreated and provided avenues to spread throughout the remainder of the ship. In order to prevent catastrophic loss to both equipment and crew from fire and smoke damage, fire simulation systems are integral to the design and training processes throughout the entire lifetime of any naval warship.

7.1.1 Fire and Smoke Simulator (FSSIM)

As with many other aspects of ship design, the process of evaluating various configurations of components and shipboard scenarios becomes particularly burdensome when traditional modeling techniques are employed. While computational fluid dynamics (CFD) is capable of providing high degrees of accuracy and valuable insight into the dynamics of a system, these simulations are extremely time consuming and entirely incapable of producing results suitable for real-time analysis. Traditional spreadsheet models are far more flexible and adaptable than CFD techniques and are capable of accommodating the speed requirements for real-time analysis. However, these models are generally incapable of handling detailed simulations of any significant magnitude due to the amount of manual effort required to create and maintain them [12].

The Naval Research Laboratory (NRL) and Hughes Associates, Inc. (HAI) worked together to develop a network fire model, released in March 2004, for the simulation of fire growth and smoke spread in multiple compartments, designed specifically to address the shortcomings of more traditional modeling techniques. Validated against experimental data from fire testing onboard the Navy's fire test platform, the ex-*USS Shadwell*, the product of this work, the Fire and Smoke Simulator (FSSIM), enables real-time monitoring of fire-based scenarios to aid in the improvement of shipboard fire suppression systems, as well the development of guidelines for personnel training of first-responders [8].

However, like the DTMS Framework, the FSSIM software system was designed specifically to address the needs of computational speed and system-level accuracy for shipboard simulations, without coupling the computational elements with a graphical user interface. Interfacing with the FSSIM software involves the creation and maintenance of a detailed text-based input file composed of numerous simulation data, including compartment geometries with positions and orientations of bulkheads and doorways, ventilation and fire suppression systems, and material properties and initial conditions for the various shipboard components. The construction of this input file can often be tedious, time-consuming, and error-prone, typically requiring detailed working knowledge of the FSSIM software system [12].

7.1.2 FireGUI

While the FSSIM software system provides a powerful, accurate, and real-time simulation environment for the analysis of fire propagation and prevention onboard naval warships, the user-interface possessed the potential to be greatly improved using external systems that need not understand the details of the underlying modeling system. Over the past few years, the Cooperative Computing Group at Mississippi State University consisting of Dr. Tomasz Haupt, Mr. Gregory Henley, Ms. Bhargavi Parihar, and Mr. Robert Kirkland has worked with the members of NRL and HAI to develop a graphical user interface for FSSIM [12].

The resulting FireGUI software environment consists of four primary elements that each contribute to an enhanced FSSIM user experience. The first step involves a CAD processor, which uses production CAD drawings from existing ship designs to feed the geographic information into a central database. This central database also contains various material properties and system defaults that provide a complete physical representation of the naval vessel. Once all of the available ship data has been imported

to the database from external sources, the FireGUI graphical interface is responsible for presenting the user with a three-dimensional representation of the ship that permits subsequent manipulation of the system parameters. The FireGUI software then creates an input file for the FSSIM software system based on the CAD representation of the ship and required user inputs, which allows the FSSIM software to execute a fire and smoke simulation based on the actual ship layout. In addition, the results of the FSSIM simulation are presented within the FireGUI graphical interface in real-time, using user-customized data presentations and color schemes. The overall simulation environment is shown in Figure 7.1:

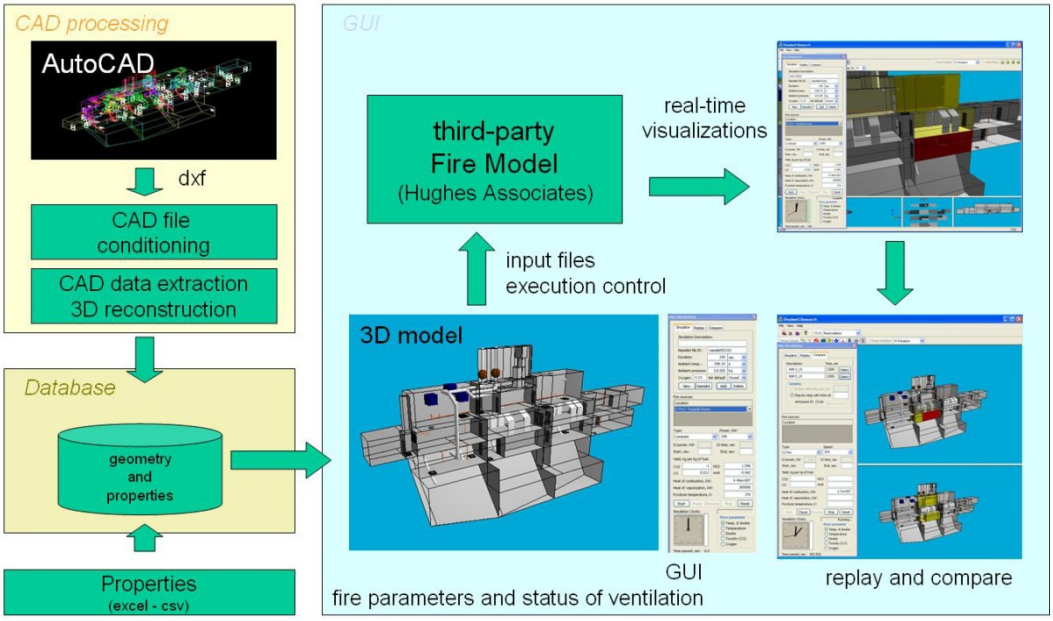


Figure 7.1: Simulation environment for fire and smoke simulations [12]

The most important user aspect of this system is the FireGUI graphical interface which directly interacts with the three-dimensional (3-D) ship representation, customizes the system parameters, and graphically presents the results of a simulation. Throughout the simulation process, the user may interactively examine the geometry of the ship with

numerous customization options, including a 3-D display of the entire ship geometry (Figure 7.2A); rotational, translational, and zooming capabilities (Figure 7.2B); selective suppression of entities such as doors and hatches (Figure 7.2C); separation of shipping decks to view interior compartments (Figure 7.2D); selective suppression of individual compartments and entire decks (Figure 7.2E); wireframe views to focus on the fire suppression system and smoke propagation (Figure 7.2F); a floor plan view to allow overhead examination of compartment layouts (Figure 7.2G); and 3-D views of individual compartments (Figure 7.2H):

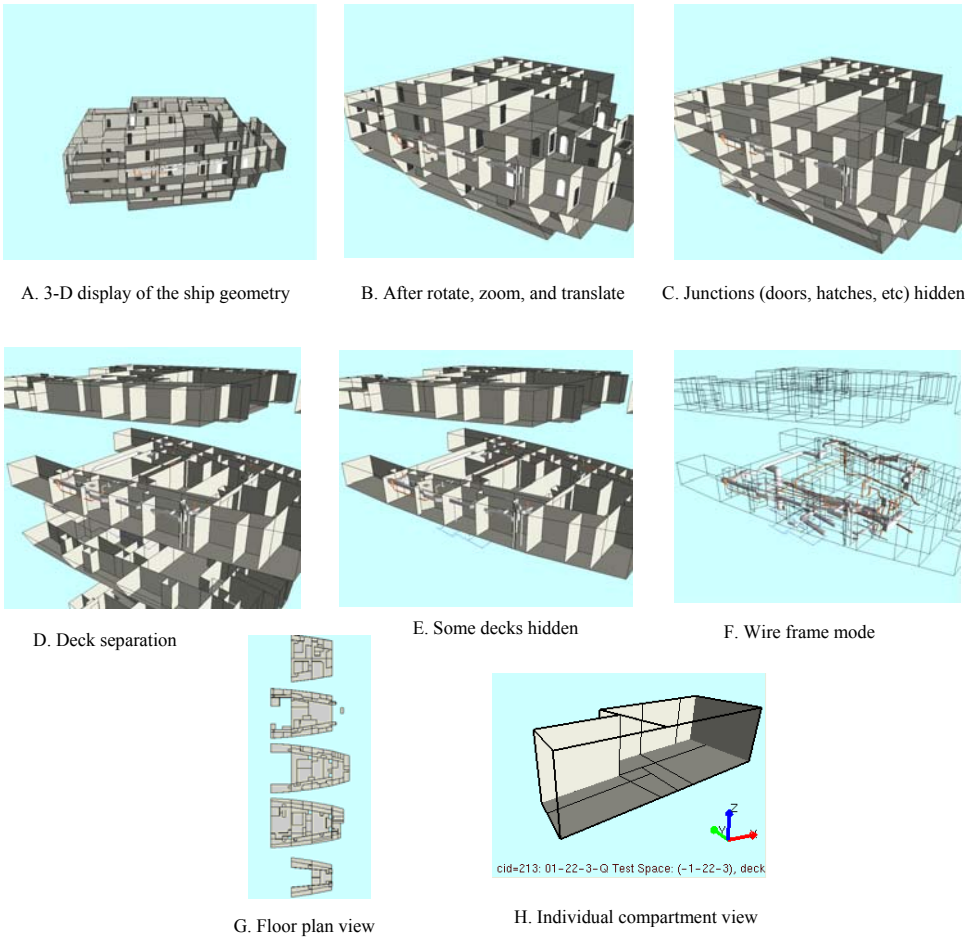


Figure 7.2: Interactive capabilities of FireGUI [12]

Any object within the FireGUI graphical environment may be selected in order to view and modify its properties. Custom dialog boxes are displayed based on the type of object selected; for example, compartment walls will have properties that differ from aspects of the ventilation system. Examples of the property dialog boxes are shown in Figure 7.3.

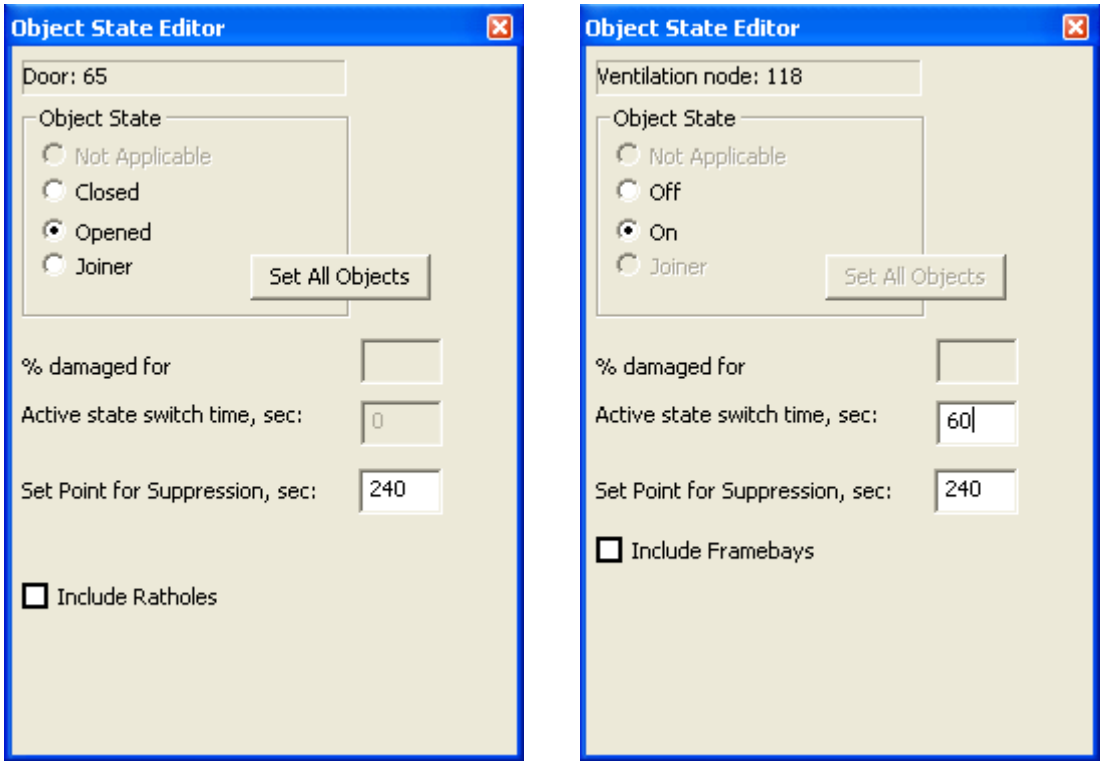


Figure 7.3: Property dialog boxes in FireGUI [12]

Once the simulation has been completely parameterized, FireGUI generates the necessary input file for the FSSIM software system, and the results of the simulation are viewed graphically in FireGUI as they are calculated in real-time by FSSIM. Any of the interactive views shown in Figure 7.2 are available during the presentation of the results, thus allowing the user full interaction with the ship geometry. Furthermore, the results of the simulation are stored externally, thus allowing a visual presentation to be replayed as

necessary without the need to recompute the results through the FSSIM software. Similarly, the results may be exported in a common file format to allow for more in-depth study using external analysis tools. For example, Figure 7.4 shows comparative results of oxygen depletion from two separate simulations within the FireGUI interface:

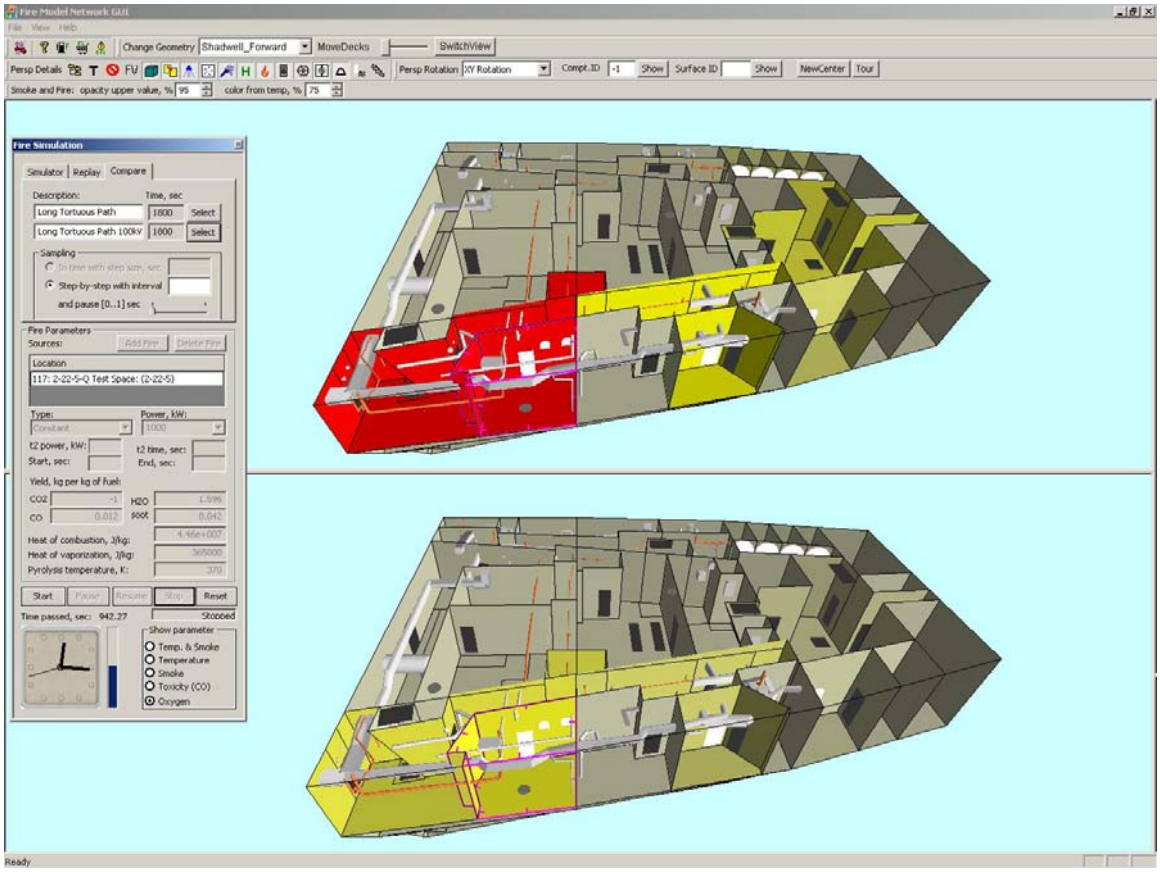


Figure 7.4: Comparative simulation results in FireGUI [12]

The FireGUI graphical interface clearly brings immense power to the FSSIM software system. The CAD processor allows users to avoid the tedious task of recreating each component of the ship in the format preferred by the FSSIM software, and it allows much more complex and sophisticated systems to be simulated while simultaneously reducing the required development time. The interactive 3-D environment provides a

greater understanding and appreciation of the ship layout, the nature of its compartments, and the various subsystems that run throughout. Finally, graphical presentation of the results provides far more insight into the complex interactions of numerous shipboard systems than would ever be possible with raw numerical data.

7.2 DTMS-FIREGUI INTEGRATION

The DTMS Framework has been specifically designed to provide a simulation environment that very closely resembles the structure provided by the FSSIM software system. Based on a network modeling system, the focus of the development effort behind the DTMS Framework has been to concentrate on speed, accuracy, and flexibility. The DTMS Framework is not tied to any particular graphical interface or results analysis tool and has been provided with a sophisticated interfacial system that allows it to be easily coupled with external systems to provide a richer user experience.

In sharing many similarities with the FSSIM software system, the DTMS Framework was recognized as ideally suited for integration with the FireGUI graphical interface. Since both FSSIM and DTMS are primarily designed to be used in naval applications, much of the system architecture that has been designed into FireGUI is directly applicable to the DTMS Framework. Minimal changes need be made to either the DTMS Framework or the FireGUI interface to allow interaction between the two software systems. Specifically, the CAD processor for the FireGUI interface must be able to recognize the physical components of the thermal management system such as chillers, piping, air handling, and ventilation systems, rather than the fire suppression systems utilized in the FSSIM integration, and the FireGUI interface must additionally provide a means of visualizing these components inside the graphical environment. The DTMS Framework must in turn provide a means of processing the text-based input files

produced by the FireGUI interface and initiating a dynamic simulation from the input provided.

The following sections present the development of a text-based input system to interface the text files used by the FireGUI graphical interface with the input system designed for the DTMS Framework which was discussed in Chapter 5.

7.2.1 DTMS Input Deck

In order for FireGUI and the DTMS Framework to work together, a common mechanism has been designed to allow the two software systems to communicate effectively. Building on previous work performed by the Mississippi State University team, a text-based communication system has been implemented within each of the software programs based on the `namelist` I/O feature that was added to the Fortran 90 language standard and is capable of being read and written by both DTMS and FireGUI.

The Fortran `namelist` mechanism consists of a sequence of records or *cards*, each of which contains a group of variables and their values that relate to a particular object model. A sample `namelist` record is provided below:

```
&namelist-group-name var1=x, var2=y, var3=z
```

Figure 7.5: Sample namelist record

The line begins with the `&` sign, followed immediately by the name of the data group. For DTMS-FireGUI purposes, there are five primary group names and several additional group names that are used to communicate with the two systems. The primary group names correspond to the five primary base classes found in DTMS: `MODEL`, `CONTROL`, `FLUID`, `SOLVER`, and `SIMULATION`, while the additional names correspond to certain model specializations: `FLOWMODEL`, `EFFORTMODEL`, `GROUPMODEL`, and `DTMSMODEL`.

The group name is followed by a comma-separated list of the data that corresponds to the current group. Each piece of data is represented by the name of the variable followed by an equal sign and the data value.

```
&SIMULATION outputFileName='SampleInputDeck.csv', finalTime=900,
timeStep=1, writeStep=1

&SOLVER id=1, type='NewtonRaphsonResistiveSolver',
errorTolerance=0.0001

&FLUID id=2, type='Water', pressure=202650.0, enthalpy=28.146

&EFFORTMODEL id=3, name='Source', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=2,
pressure=820995.0, enthalpy=43.8783,
writeVariables='pressure:P','enthalpy:h','temperature:T'

&FLOWMODEL id=4, name='CoolantPipe', type='Pipe', dependent=.true.,
output=.true., solverID=1, effortIDs=3,5, fluidID=2,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headLength=0.0,
writeVariables='flow:W','flowCoefficient:C','enthalpy:h','temperature
:T'

&EFFORTMODEL id=5, name='Sink', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=2,
pressure=203303.0, enthalpy=28.146,
writeVariables='pressure:P','enthalpy:h','temperature:T'
```

Figure 7.6: Sample DTMS input deck

A DTMS Simulation can now be constructed by creating a collection of separate input cards, called the *input deck*, for each model, control, solver, and fluid in the simulation. The `namelist` data format provides a simple mechanism that allows complete control over the creation and manipulation of a DTMS simulation. Additionally, this technique requires minimal effort to create a parser within other environments, thus making it ideal for promoting external integration efforts with the DTMS Framework. The human-readable nature of this input data format also allows for manual creation, examination, and manipulation of a DTMS simulation without requiring

any prior knowledge of C++. Figure 7.6 presents an example of a simple input deck for use with the DTMS-FireGUI collaboration.

The complete specification for the DTMS input deck designed to communicate between the DTMS Framework and the FireGUI graphical interface is provided in Appendix D.

7.2.2 Input Deck Parser for the DTMS Framework

To encourage modular development practices throughout the DTMS Framework, the processing system for the input deck has been separated into two distinct software components, each specializing in one aspect of the interfacial system. The first component is responsible for parsing the text-based input file generated by the FireGUI interface and storing the input data into proper C++ data structures.

To facilitate this process, a new class has been created to store the information from a single input card which has been suitably named `InputCard`. The structure of this new class is very similar to the `DTMSIOObject` classes that are described as part of the input system in Chapter 5. Each `InputCard` class contains a string variable to hold the group name that is used to identify the card and an STL map that contains the DTMS component data, stored in a string format, associated with the current card. The input cards found in the input file are read and stored into unique instances of the `InputCard` class and stored collectively inside a C++ vector object.

Due to the modular nature of the input system for the FireGUI input data file, the input deck parser only needs to examine the input file for the syntax and structure of the generalized input deck structure. The parser is not required to enforce any DTMS-specific requirements on the information from the input file, such as proper model connections, required initialization parameters, or consistency among C++ data types.

7.2.3 Conversion from `InputCard` to `DTMSIOObject`

The second component of the input system is responsible for converting the input card data structures into the `DTMSIOObject` subclasses that are required by the input system described in Chapter 5. The group name variable that is stored inside each `InputCard` class is used to select the proper `DTMSIOObject` subclass that will be used to store the DTMS data. Once this information is available, the system locates and extracts the `type` and `ID` parameters from the `InputCard` objects to store in the corresponding `DTMSIOObject` objects. The remaining data members stored in the STL map of each `InputCard` object are then copied into the map data structure of the `DTMSIOObject` subclass object.

After the `DTMSIOObject` subclasses have been created and initialized using the data from the `InputCard` data structures, the remaining aspects of the DTMS input system are utilized to complete the simulation initialization and invocation process. The `type` variable found in each `DTMSIOObject` is fed to the various factory classes used by the DTMS Framework to generate the corresponding DTMS components, which are then initialized by calling each `loadState` function with the necessary `DTMSIOObject`.

7.3 DEMONSTRATION OF FIREGUI INTEGRATION

Preliminary efforts were begun to integrate the FireGUI graphical environment with the DTMS Framework backend by using the `namelist` communication mechanism. Using a pre-constructed sample simulation system, the FireGUI environment is capable of visualizing the layout of the system, manipulating the parameters of the DTMS models, initializing the DTMS simulation, and visualizing the results that are returned by the DTMS Framework.

To demonstrate results of the collaboration efforts between the two universities to integrate these two tools, a simple example simulation is constructed in this section using

the FireGUI environment and then executed using the DTMS Framework. The system that is represented is a heavily-simplified representation of a vapor-compression water chiller as found onboard the DDG-51 guided missile destroyer. A complete simulation of this chiller system is modeled in the thesis of Patrick Hewlett [13]. However, for the purposes of this example, the physical behaviors of the components are heavily simplified such that the resulting simulation only minimally reflects the features of the physical system after which it is modeled. Regardless, this system represents a meaningful demonstration of the capabilities provided via integration of the DTMS Framework and the FireGUI interface by capturing the primary physical behaviors and performance characteristics that are found in every DTMS simulation.

Figure 7.7 below shows the complete model of the York 200-ton chiller constructed for the DTMS Framework [13]:

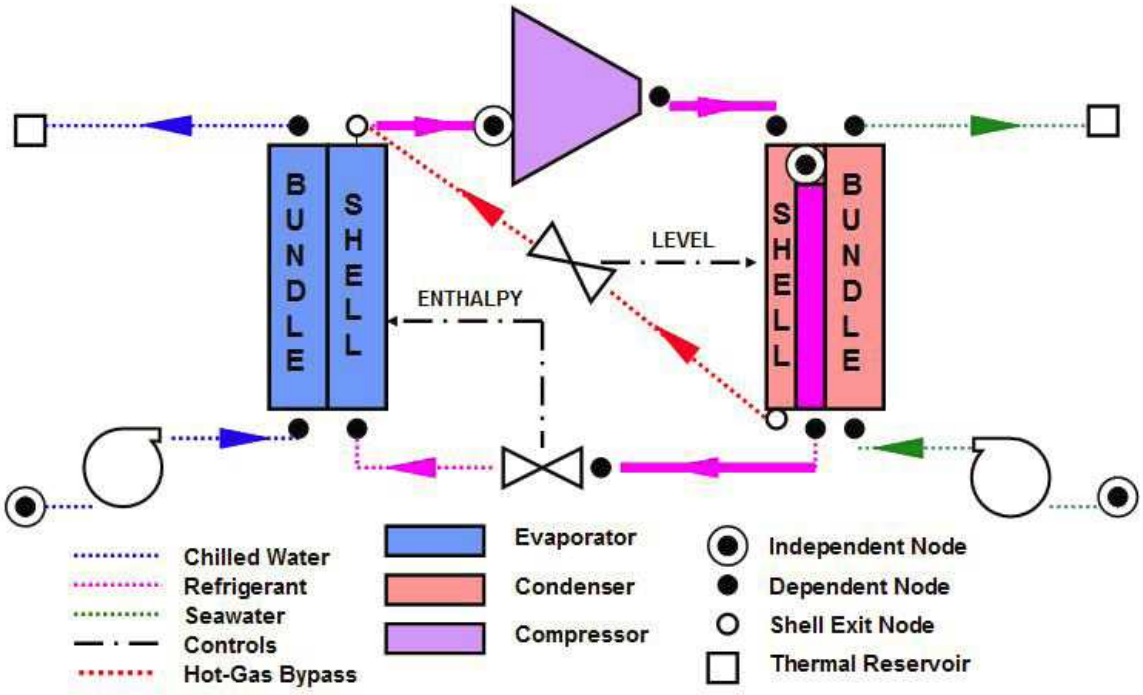


Figure 7.7: Complete DTMS representation of the York 200-ton chiller

This figure presents the major components of a typical two-phase chiller, along with the controls utilized to maintain proper performance, and the coolant loops to which the chiller is connected. The left side of the diagram shows an abstraction for a chilled water loop that is used onboard the DDG-51 to transfer cooled water to the various thermal loads throughout the ship, while the right side presents a seawater loop that transfers excess heat away from the chiller for discard into the ocean. The refrigerant loop in the center of the diagram represents the primary vapor-compression chiller components that are responsible for transferring heat from the chilled water loop to the seawater loop.

In this demonstration, a simplification of the chilled water loop and the refrigerant loop are modeled using the FireGUI graphical interface. The central refrigerant loop consists of four major components: an evaporator, compressor, condenser, and expansion valve. While the seawater loop has been omitted for simplicity, the behavior of this loop has been captured by directly modifying the performance characteristics of the condenser model. The chilled water loop is modeled using a pair of heat exchangers that each supplies a fixed heat load to the chilled water system. Figure 7.8 below presents a simplified version of the complete chiller model shown in Figure 7.7 and focuses on the primary components that are utilized in the following demonstration.

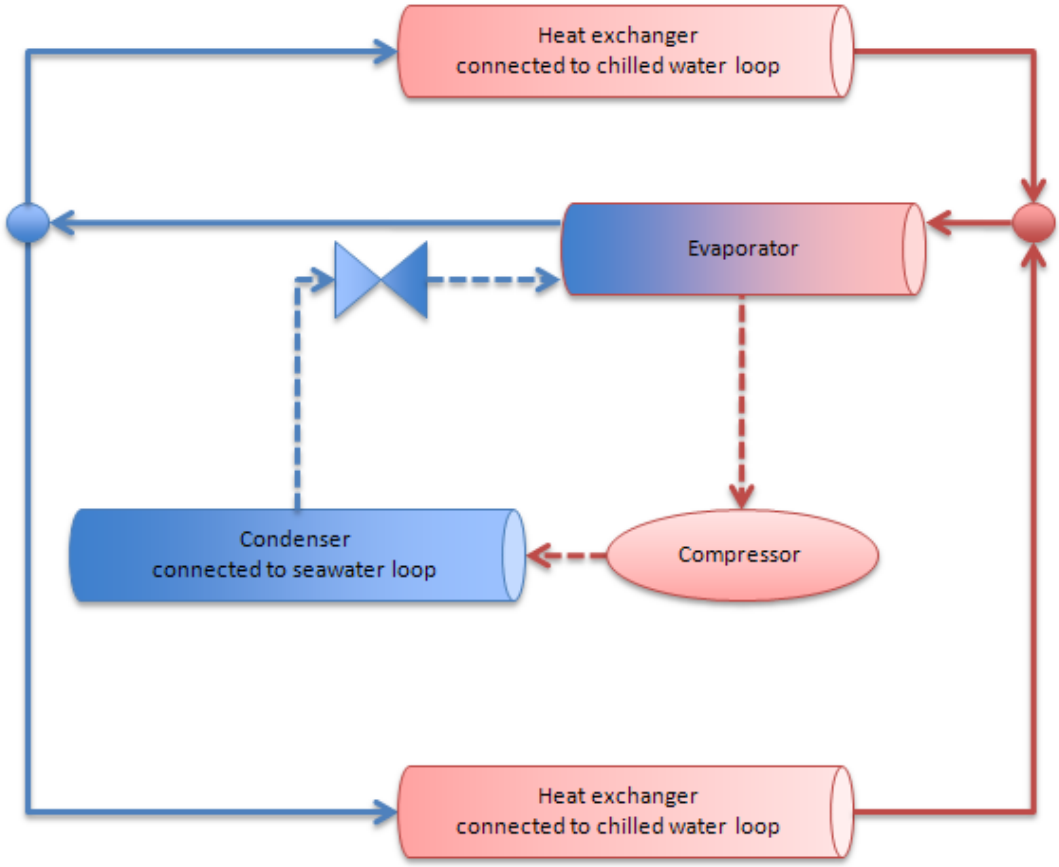


Figure 7.8: Component diagram of the two-phase chiller model

While the complete chiller model operates using multiple fluids with two-phase characteristics, the chiller system modeled in this demonstration has been reduced to a single-phase, water-based coolant loop. By using a single working fluid, the inner refrigerant loop is directly combined with the chilled water loop, which allows for several of the components to be replaced with simpler representations. Since the evaporator is no longer responsible for transferring heat between two separate fluid loops, it is replaced with a simple mixing tank to combine the hot and cold flows of the system. The condenser is replaced with a single-phase heat exchanger to interact with the seawater loop, and the compressor is represented by a centrifugal pump since the fluid remains in

the liquid phase throughout the coolant loop. Figure 7.9 then reflects the simplifications that have been made to the chiller system for the purposes of this demonstration:

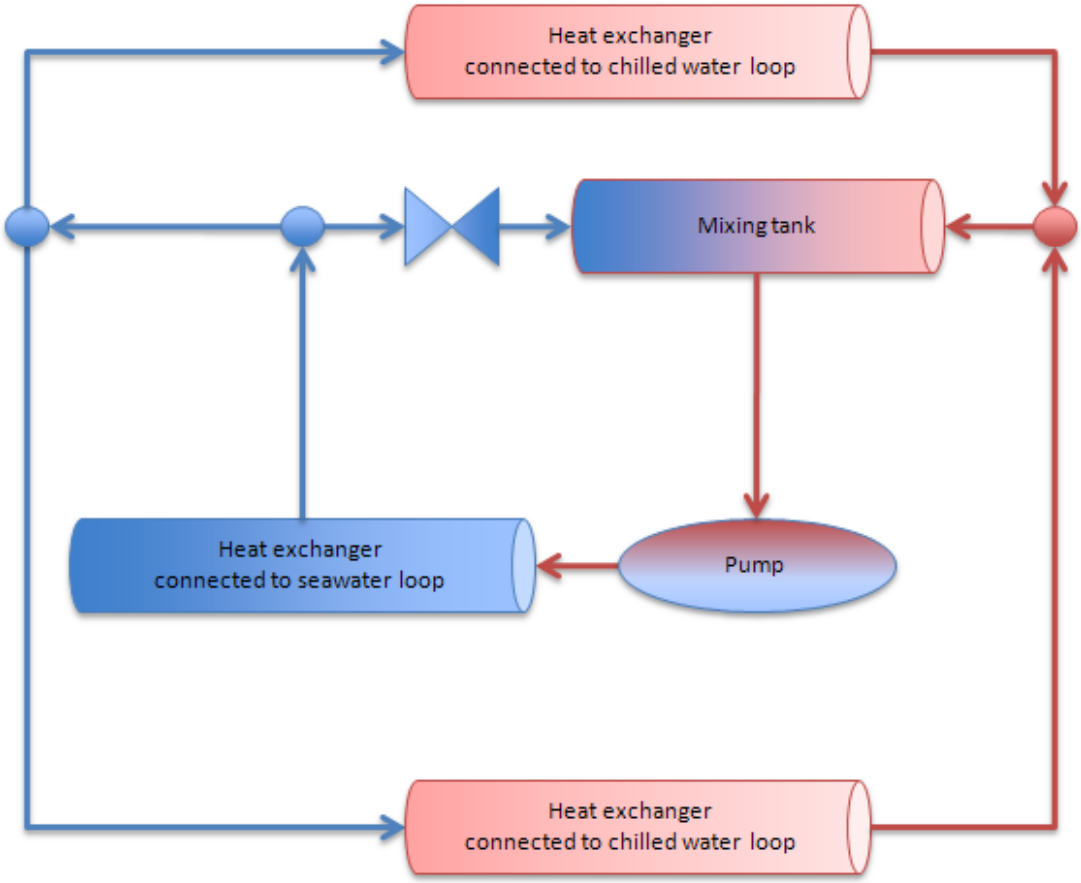


Figure 7.9: Simplified version of the DTMS chiller model

The seawater heat exchanger is modeled as a source of constant pressure and enthalpy within this simple system, and the simulation solvers perform more efficiently when the pressure source is placed upstream of the centrifugal pump. Thus, to make the modeling process more efficient inside the DTMS Framework, the location of the pump and the seawater heat exchanger are reversed. Figure 7.10 depicts the final component diagram

of the heavily simplified chiller system that is modeled using the FireGUI graphical interface and the DTMS Framework.

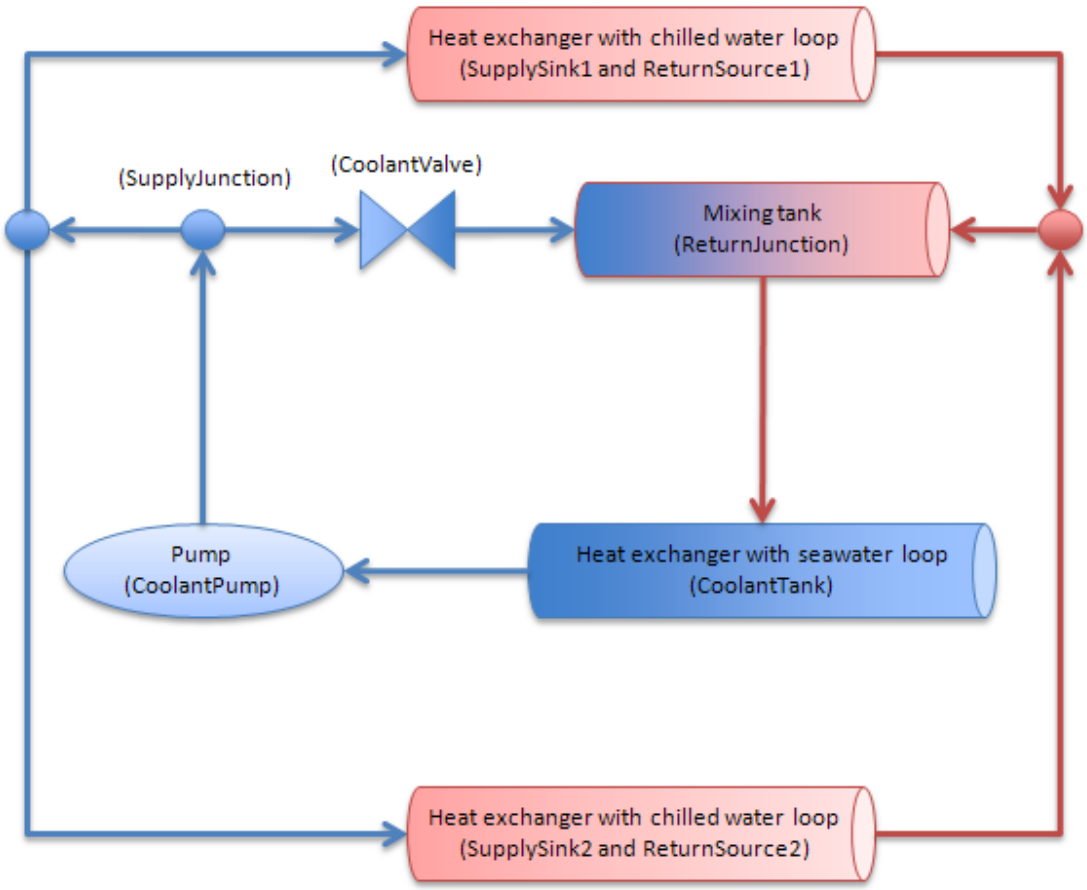


Figure 7.10: Component diagram of the heavily simplified chiller model

This diagram additionally contains the names of the various components that are used to model the system in the DTMS Framework. Figure 7.11 shows each of the models that are present in the complete DTMS representation of this simplified chiller system. The final DTMS model contains all of the piping structure and the internal pressure nodes that are required to completely connect the sub-elements of this system.

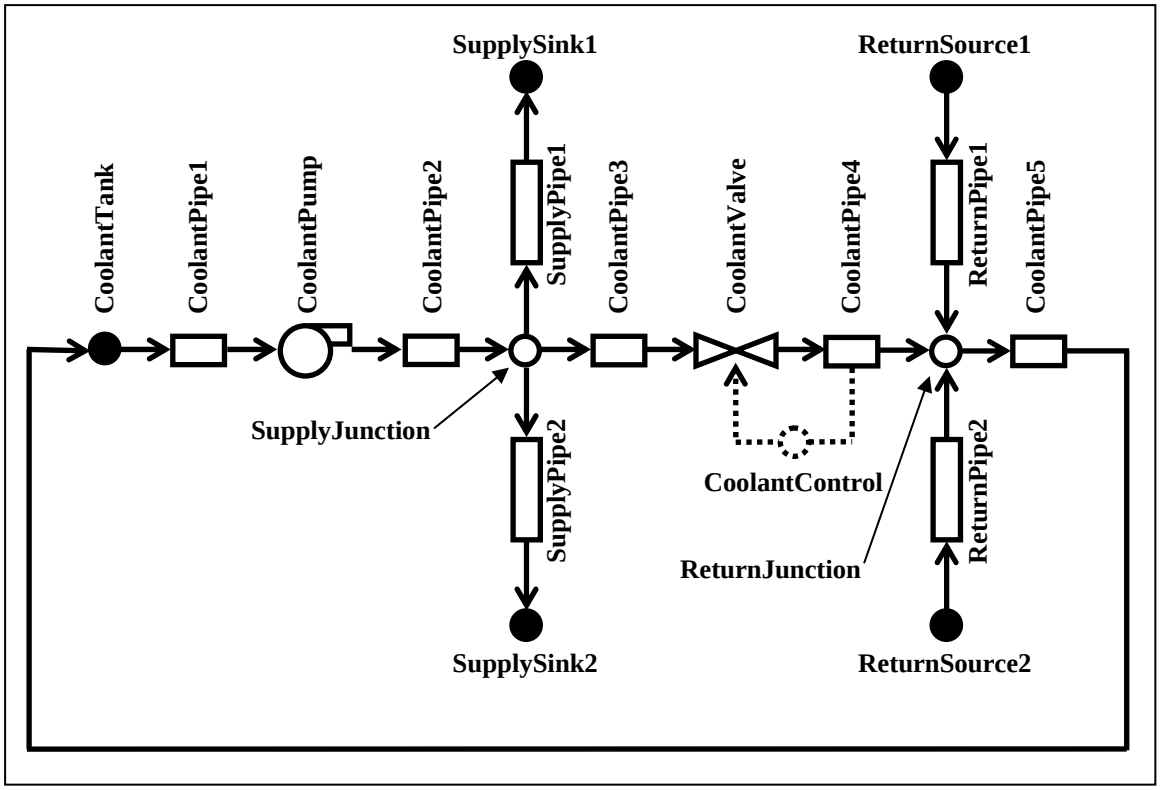


Figure 7.11: DTMS components used to model the simplified chiller system

Although the naming conventions are different from previous figures, each of the primary component models from the coolant system shown in Figure 7.10 is found in Figure 7.11. The mixing tank is represented with a `ThermalFluidEffortModel` named `ReturnJunction`, the seawater heat exchanger is represented with a `ThermalReservoir` named `CoolantTank`, and the `CentrifugalPump` model `CoolantPump` represents the pump. A complete demonstration documenting the construction of this simulation within the DTMS Framework is provided in Appendix C.

The system described above has been reproduced using the FireGUI graphical interface in order to provide a means of initializing the various components of the system and visualizing the results of the simulation. Figure 7.12 below shows the coolant loop within the FireGUI environment with each of the various DTMS components represented

in the system. In this particular view, several of the walls of the surrounding compartments have also been included to indicate spatial separation between the component models, a key feature that FireGUI introduces in this collaboration.

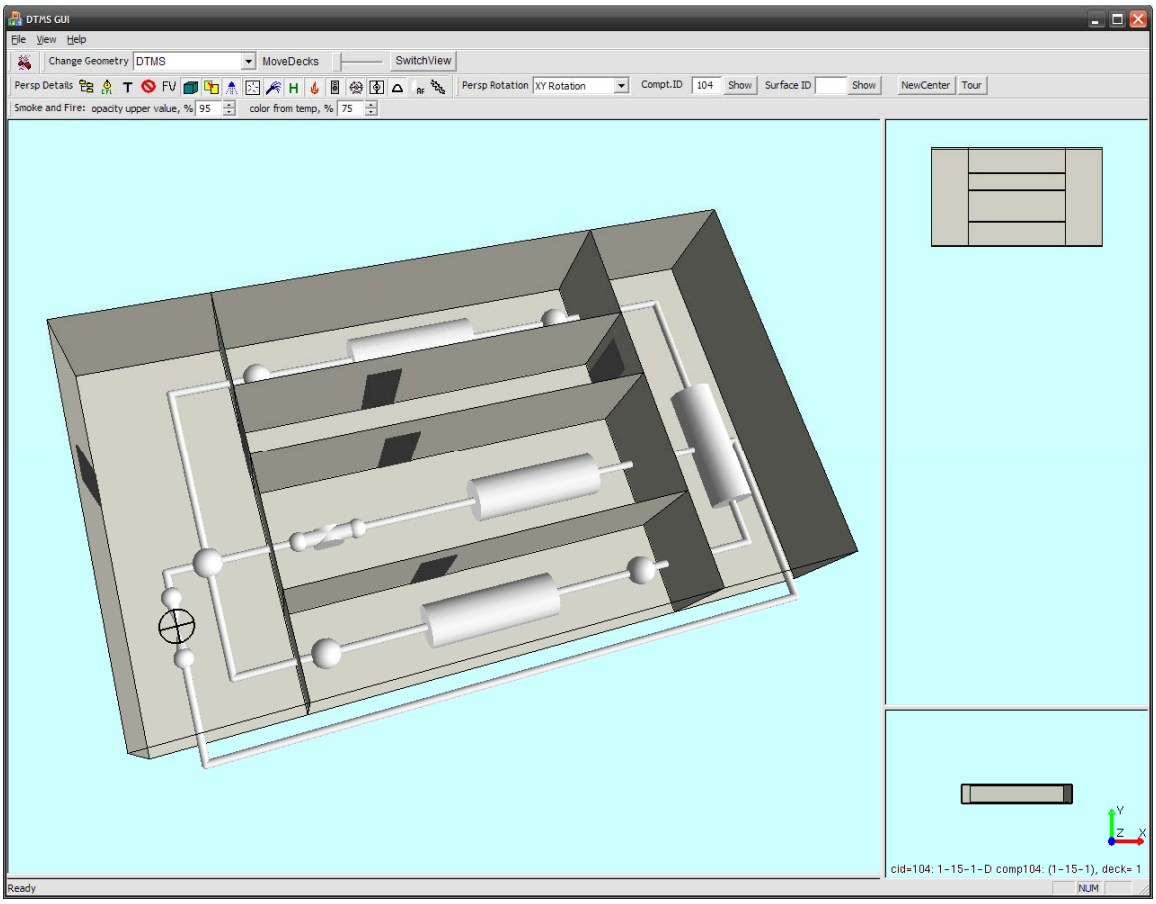


Figure 7.12: Coolant loop in FireGUI

Using only the FireGUI environment, all of the parameters of the associated DTMS models may be manipulated and modified by the user. For example, the following figure shows the data screen for one of the pipes found in the coolant loop:

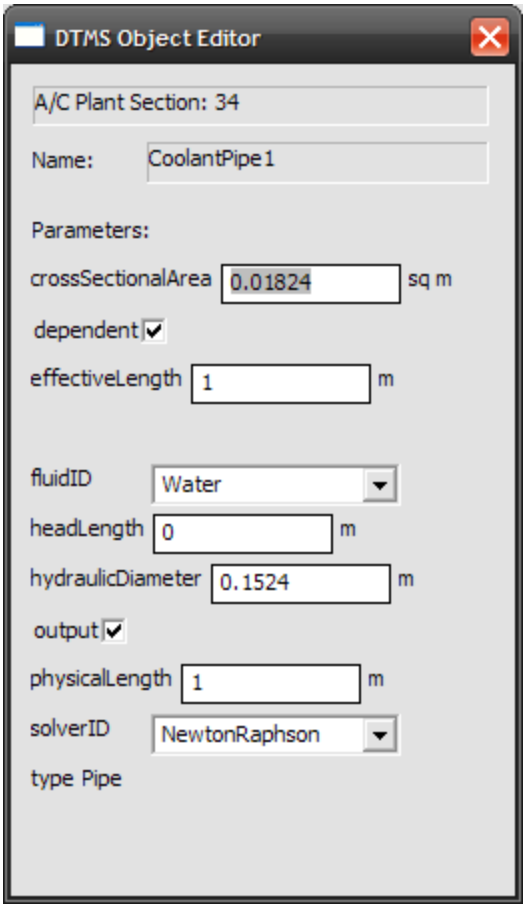


Figure 7.13: Pipe model data in the object editor

Once all of the models have been properly parameterized and initialized by the user, the DTMS simulation may be initiated directly from within the FireGUI environment. The figure below shows the simulation dialog box, which allows the user to set the name of the output file, the length of the simulation, and the time step for the simulation:

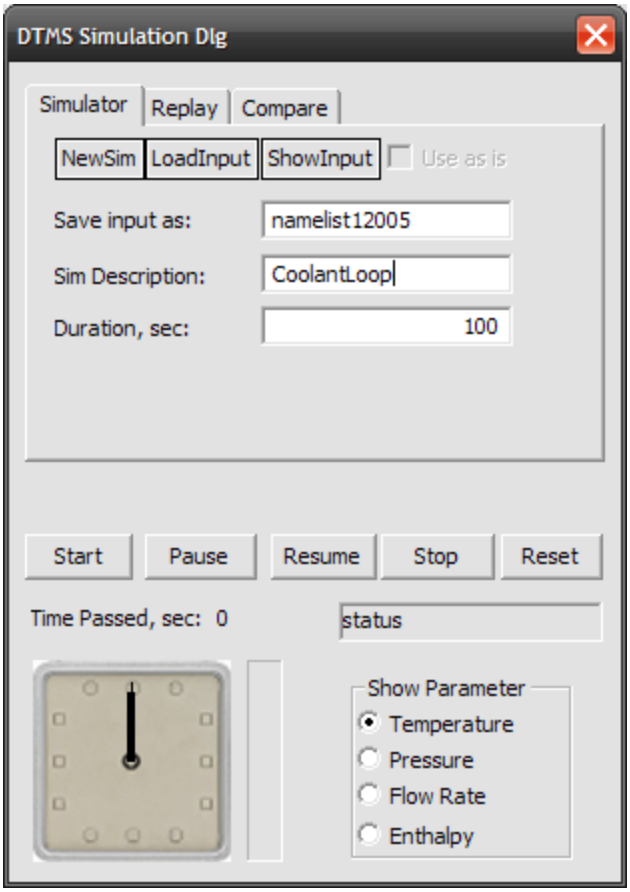


Figure 7.14: Simulation editor

While a simulation is running, the results from the DTMS Framework are continuously sent back to the FireGUI environment, allowing the user to visually monitor the simulation as it progresses. The following figure shows the simulation of the simple coolant loop in FireGUI while the DTMS Framework is calculating the results for the system in the background. In this view, the compartment walls have been removed to allow easier viewing of the temperatures within the various component models.

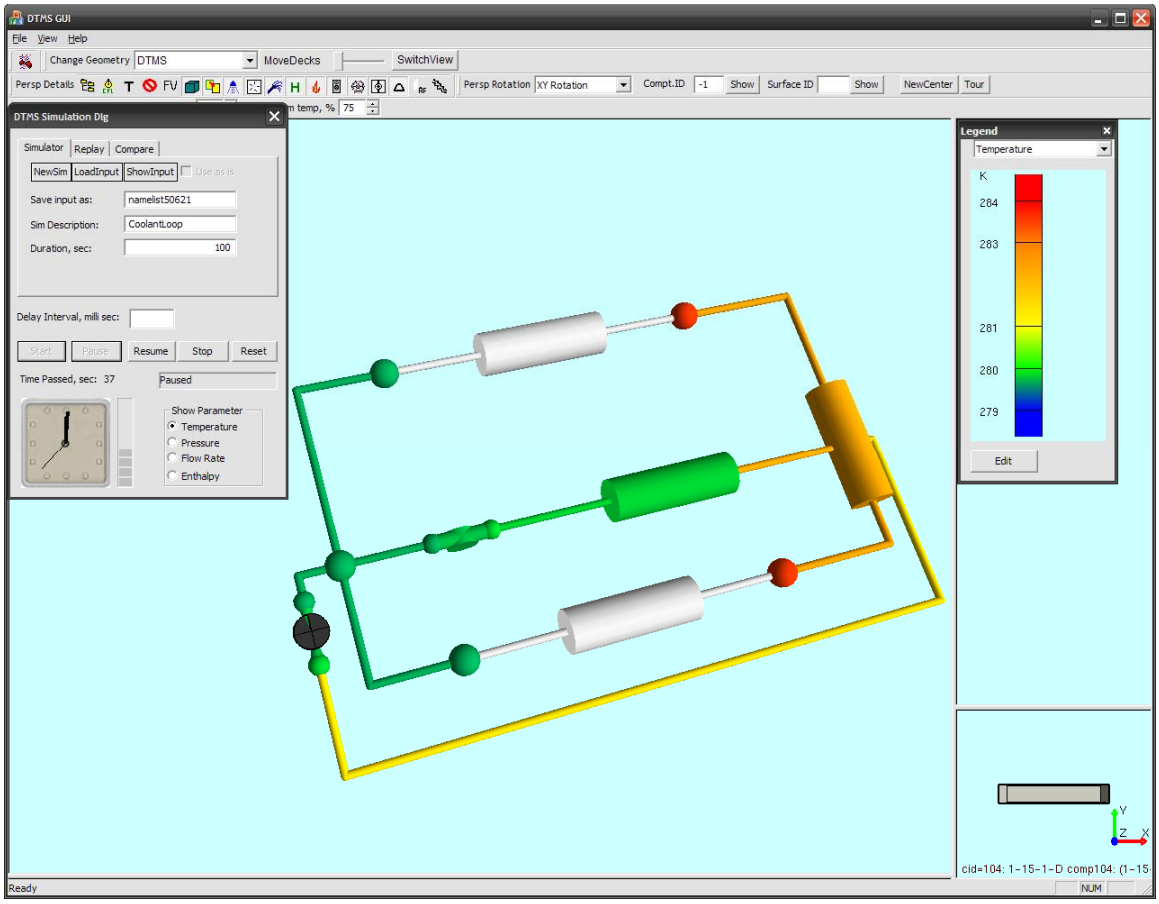


Figure 7.15: Coolant loop simulation showing temperature distributions

From the FireGUI interface, the distributions of temperature, pressure, mass flow rate, and enthalpy throughout the simulated system may each be viewed in a dynamic sense while the DTMS simulation is executing. Figure 7.15 shows the resulting discrete temperature gradients between the components in the chiller system, while Figures 7.16, 7.17, and 7.18 show pressure, mass flow rate, and enthalpy distributions, respectively.

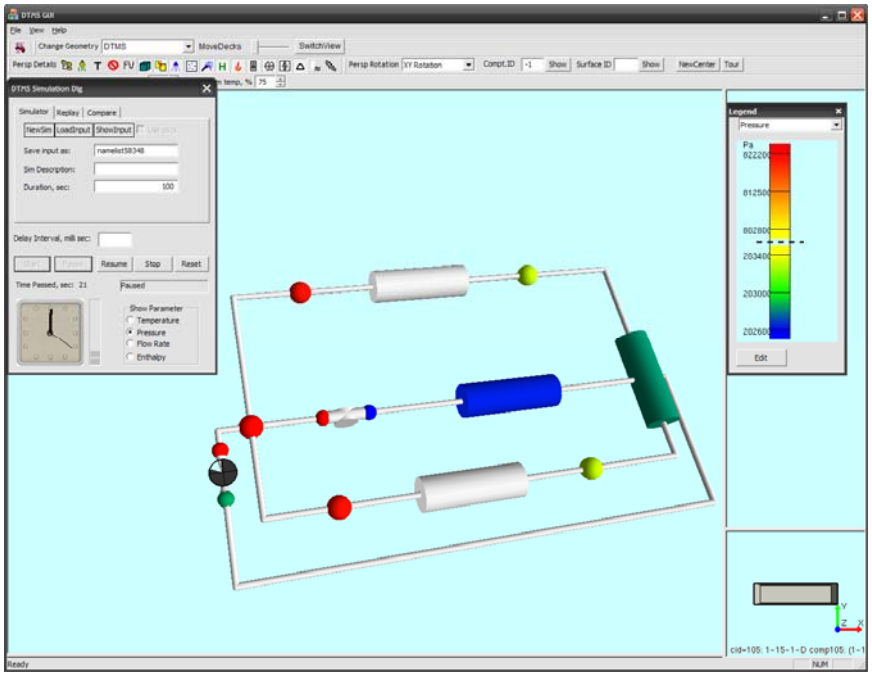


Figure 7.16: Coolant loop simulation showing pressures at the effort models

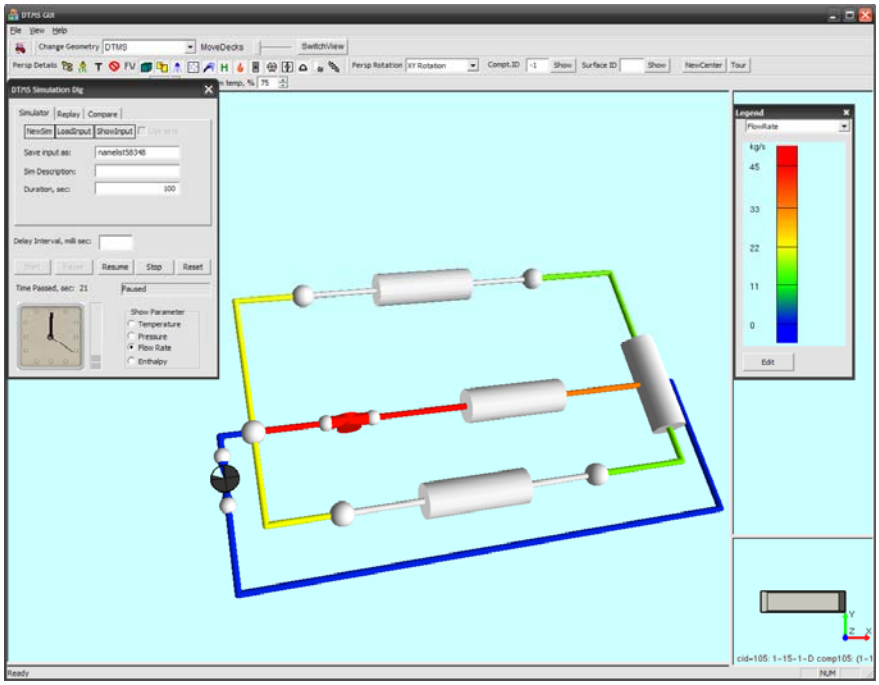


Figure 7.17: Coolant loop simulation showing flow rates through the flow models

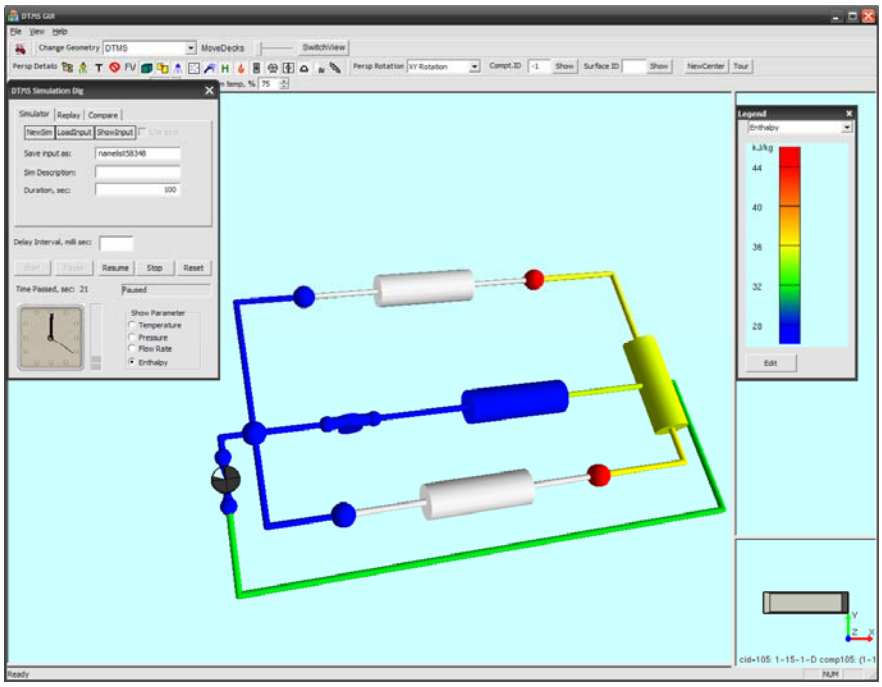


Figure 7.18: Coolant loop simulation showing enthalpy distributions

While the integration of the FireGUI graphical environment and the DTMS simulation framework is clearly in its infancy, significant progress has been made toward realizing its full potential. Each system has benefited greatly from the interaction with the other, and additional work is currently being pursued at both universities to enhance the interaction between these two software systems.

Chapter 8: Conclusions and Recommendations for Future Research

8.1 CONCLUSIONS

Over the past few years, development of the DTMS Framework has produced the architecture for a complex, system-level modeling and simulation environment specifically designed to address issues of thermal management onboard an AES. Compared with commercial software, DTMS offers a more accessible, flexible, and customizable framework for dynamic, system-level modeling of any system that can be represented using bond graph theory. The work of this thesis has sought to increase the simulation potential and usability of the DTMS Framework as the software of choice in the simulation future of the ESRDC, the US Navy, and commercial shipbuilders.

The introduction of a modeling architecture for inertial and capacitive physical behaviors in the DTMS Framework has greatly increased the potential applicability of the framework to the complex, interconnected, and multi-disciplined simulation challenges invovled in the AES. While previous work using the DTMS Framework concentrated primarily on thermal fluid systems, the enhancements presented in Chapter 4 have provided the tools necessary for simulation of sophisticated electrical and mechanical systems. The thermal-electrical co-simulation of the power grid conversion systems presented by Matthew Pruske [23] has relied heavily on the inertial and capacitive representations presented here, which has in turn allowed complex interactions between the various sub-elements of any modern naval architecture to be realized in a manner previously unavailable.

With the focus of the design work of the DTMS Framework centered around the speed, accuracy, and systems applicability of the modeling and simulation environment, the Spartan, albeit utilitarian, user interface of the DTMS Framework has provided a

gateway to the future of thermal systems modeling. The interfacial system presented in Chapter 5 provides new avenues for the integration of the DTMS Framework with external software systems that allow for an improved user experience and virtually unlimited application potential. Additions to the DTMS output system, along with the addition of sophisticated debugging and error handling systems within the DTMS Framework presented in Chapter 6, provide numerous methods for the extraction of meaningful data from the results of a DTMS simulation. This architecture further provides modeling development tools to aid DTMS developers with the design and implementation of future thermal models and simulations.

The collaboration work with Mississippi State University to provide a bridge between the DTMS Framework and the FireGUI graphical interface has sought to demonstrate the enormous power provided by each of these software systems for complex, system-level simulations. The initial results of these collaborative efforts presented in Chapter 7 show the beginnings of a fruitful relationship that has the potential to provide new and meaningful representations of shipboard simulation data, as well as far more insight into the complex interactions of numerous shipboard systems than would ever be possible with raw numerical data.

8.2 RECOMMENDATIONS FOR FUTURE RESEARCH

While the work presented in this thesis has greatly increased the power and usefulness of the DTMS Framework for simulating complex, interacting shipboard systems, additional work is recommended to realize the full potential of the modeling and simulation capabilities provided by this framework. The following sections present recommendations for this work wherein the theme is to improve the speed, accuracy, and applicability of the DTMS Framework as the preferred design tool for thermal simulation research performed through the ESRDC.

8.2.1 Compiled Libraries

One of the most important considerations throughout the design process of the DTMS Framework has been the protection of the intellectual property of the developers. Every portion of the framework has been specifically designed to be fully functional in a closed source code environment where the developer does not have, and does not need, direct access to any part of the DTMS Framework created by others. The development of the integration between the DTMS Framework and the FireGUI graphical interface has been handled through a completely divided development effort, where neither university requires any source code from the other.

This has been accomplished through the use of static libraries for much of the C++ code used to power the DTMS Framework. Static libraries allow portions of the framework to be compiled separately to protect the content of the source code, and each of the separate pieces are eventually linked together to form a complete, working simulation executable. While this process has been suitable for the continued development the DTMS Framework thus far, the process of creating the static libraries can be a tedious and time-consuming task. In order to be properly linked together, all of the static libraries used for a project must be compiled using precisely the same compiler configuration. Therefore, all DTMS developers must either use the same development environment, which limits the applicability of the framework for multiple environments, or the core aspects of the framework must be recompiled for each of the numerous possible compilers on multiple platforms.

This limitation of static libraries is commonly overcome through the use of dynamic link libraries (DLLs). These libraries require separate portions of a program to communicate through a standard interface, but they are linked with external programs at run time, rather than compile time, and do not require that the separate pieces of the

framework utilize the same development environment. These libraries must only be compiled once for a particular platform and then no longer need be linked with external programs during any additional compilation processes. Although dynamic link libraries require additional development effort in order to properly define the interfaces through which the disparate parts of the DTMS Framework will communicate, it is recommended that the DTMS Framework be converted to utilize DLLs in order to take advantage of their development-friendly features.

8.2.2 Input System

While the universally-applicable input system presented in Chapter 5 provides full developer access to the creation, parameterization, and execution of a DTMS simulation, the connection of models through this interface represents a unique challenge. The object factory classes of the input system operate through the base classes of the DTMS Framework, relying on the polymorphic behavior of the `loadState` functions to provide the customized initialization behavior required by each derived class.

However, the connections that are maintained between the models and solvers of the DTMS Framework require knowledge of certain derived class behaviors. For example, the `ResistiveNetworkFlowModel` class must have upstream and downstream connections to the `ResistiveNetworkEffortModel` objects surrounding it, and the `ResistiveNetworkSolver` classes must be able to differentiate between the `ResistiveNetworkFlowModel` and the `ResistiveNetworkEffortModel` classes in order to properly construct the matrices used to resolve the flow network. The principles of polymorphic behavior prevent the characteristics of these derived class models from being accessed through the base class interfaces of the DTMS Framework.

Currently, this limitation is overcome through the use of additional object factories for the classes that are associated with the Resistive Network modeling strategy.

However, this approach requires extra care on the part of the model developer to ensure that the proper class factory is utilized for each new model developed for the DTMS Framework, and adds additional development challenges for the creation of additional modeling strategies to the DTMS Framework.

One possible method for overcoming this limitation is with the use of a C++ concept known as run-time type identification (RTTI), which allows the identification information for a derived class object to persist even when the object is referenced through a base class pointer. However, the RTTI system produces additional functional overhead in the resulting compiled program, which causes a decrease in the overall speed of the application. For this reason, nearly every common C++ compiler excludes the RTTI system from C++ executables by default.

For the connections required by the input system of the DTMS Framework, further investigation is recommended into alternative methods that would provide all of the necessary behaviors currently utilized by the Resistive Network modeling strategy while reducing the reliance on additional object factories.

8.2.3 Output System

As discussed in Chapter 6, the format specification for the CSV file format has been hard-coded into the simulation executive, which currently prevents the direct use of customized output formats for the simulation data produced by the DTMS Framework. It is recommended that a pluggable architecture for the output file format be designed and incorporated into the framework that would allow future developers to create customized plugins containing the programming logic to create customized output formats. The use of a pluggable system would allow new output formats to be added to the DTMS Framework without requiring recompilation of other aspects of the output system, similar to the plugin system designed for the object factories.

8.2.4 Simulation Solvers

With the addition of the numerical approximation methods for the inertial and capacitive models in the DTMS Framework, the behavior and performance of these components in the presence of the system solvers utilized by the Resistive Network modeling strategy must be reexamined to identify the implications of these approximations to the overall simulation stability. It is recommended that studies be performed to examine the impact of adjustments to the solver error tolerances and simulation time steps for simulations containing both linear and nonlinear representations of the inertial and capacitive models.

Additionally, it is recommended that the logic behind the Resistive Network solver routines be investigated to determine if the behaviors of the inertial and capacitive models can be directly incorporated into the solution methods, thereby eliminating the need for the numerical approximations that are currently used and potentially increasing the stability and decreasing the runtime of simulations performed using the DTMS Framework.

8.2.5 Simulation Architecture

With the expansion of the DTMS Framework to incorporate simulations from various energy domains, the importance of variable time step behavior during the course of a simulation has become increasingly important for the efficient production of simulation results. In particular, the transient behaviors of thermal components within the DTMS Framework operate on time scales on the order of tenths of a second, whereas electrical components experience transients on the order of hundredths or even thousandths of a second. When simulations incorporate models from each of these energy domains, the total time step for the simulation must be sufficiently small to capture the electrical transients, and this results in numerous unnecessary calculations for

the thermal components. Furthermore, transient behavior is often most important during the execution of dynamic events that change the state of the simulated system. In these instances, decreased time steps allow the transients of the system to be properly simulated, while larger time steps are capable of capturing the system performance during steady-state operations.

It is recommended that a system be developed to allow each model in a DTMS simulation to provide a customized maximum time step required to capture the transients of the model's physical behavior. The simulation architecture would then be responsible for executing the various model calculations only at the recommended time step, thereby improving the efficiency of the simulation calculations. Allowing the user to customize the time step at different time points throughout the simulation would also permit the transient behavior of dynamic events to be captured while maintaining efficient system calculations during steady-state operation. The creation of intelligent time step monitoring tools could automatically adjust the simulation time step as dynamic events are executed during the course of a simulation.

8.3 CONSORTIUM

As design methodologies have shifted toward system-level modeling and simulation practices, the ESRDC has proven to be a valuable asset in the development of the AES. The challenges of the AES now span several energy domains and multiple engineering disciplines, and thus the modeling and simulation efforts must confront these challenges holistically. Collaboration among members of the consortium is crucial to the continued understanding of the dynamic nature and complex system interactions of the challenges posed by the AES. Although DTMS has proven to be an effective tool in the modeling and simulation of system-level thermal management architectures, from transient behaviors of system-wide chilling loops to the dynamic heat loads produced by

the electrical distribution networks, work must continue between the various universities of the ESRDC in order to help produce the enabling technologies for the future of the US naval fleet.

While Chapter 7 presented the initial results of the collaborative effort between the University of Texas at Austin and Mississippi State University to integrate the simulation capabilities of the DTMS Framework with the enhanced user experience of the FireGUI graphical interface, the full potential of the integration between these products has not yet been fully realized. Continuing efforts should expand upon the work presented in this thesis to eventually allow full ship simulation of the thermal management system and the associated thermal loads for existing and future ship designs.

Steps toward the realization of this goal are currently being pursued through the integration of these two tools with the database of notional ship data maintained by Florida State University. The availability of this data throughout the consortium will likely prove to be valuable asset in the continued development of the technologies of the AES, and the integration of these tools will provide a common modeling and simulation experience for members of the consortium, the US Navy, and commercial shipbuilders.

Appendix A: Nomenclature

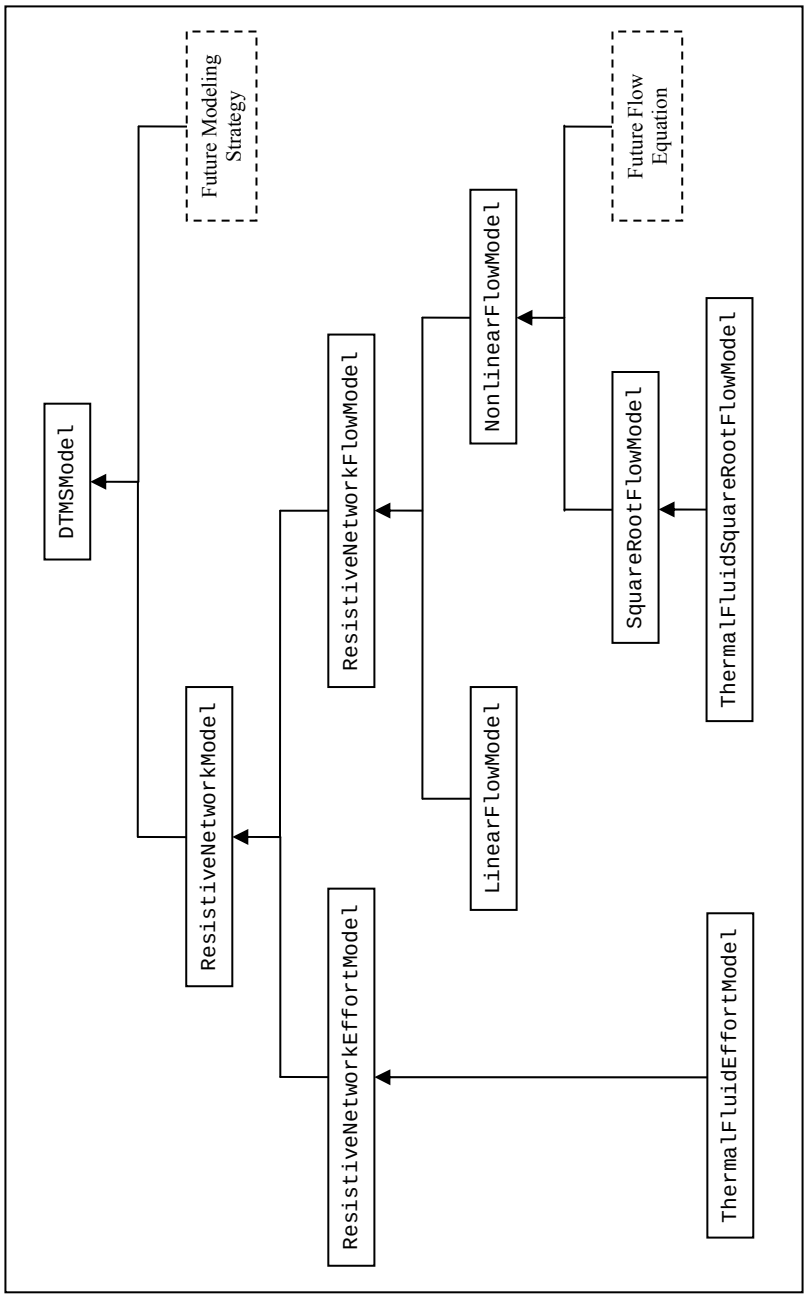
Note: Unless indicated otherwise, variables may be assumed dimensionless.

Variables

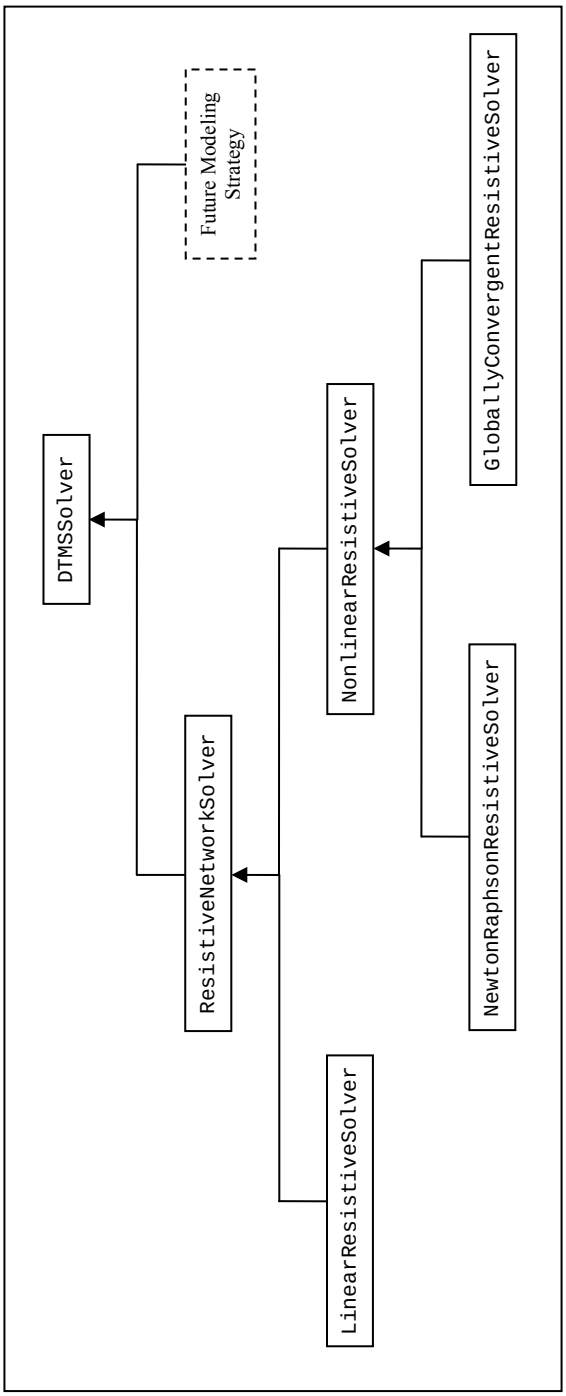
A	area	[m^2]
b	source term	
C	capacitance	[F]
C	conductance	
e	effort	
E	energy	[J]
e	specific energy	[J/kg]
f	flow	
G	conductance	
g	gravity	[m/s^2]
h	time step	[s]
i	current	[A]
I	current	[A]
I	flow	
K	design constant	
KE	kinetic energy	
L	inductance	[H]
m	mass	[kg]
$\dot{m}$	mass flow rate	[kg/s]
N	rotational speed	[rpm]
p	momentum	
P	pressure	[N/m^2]
PE	potential energy	[J]
q	charge	[C]
q	displacement	
Q	heat	[J]
Q	volumetric flow rate	[m^3/s]
R	resistance	[ohms]
S	source term	
T	temperature	[K]
t	time	[s]
U	internal energy	[J]
v	velocity	[m/s]
V	effort difference	
V	voltage	[V]
V	volume	[m^3]
W	work	[J]
w	specific work	[J/kg]
z	height	[m]
λ	magnetic flux linkage	[Vs]
ρ	density	[kg/m^3]
ϕ_R	resistive flow function	
ϕ_I	inertial flow function	
ϕ_C	capacitive flow function	

Appendix B: Class Diagrams for the DTMS Framework

Shown below is the C++ class diagram for the `DTMSModel` base class and its derived classes.



Shown below is the C++ class diagram for the `DTMSSolver` base class and its derived classes.



Appendix C: DTMS Framework User's Guide/Tutorial

Version 2.2, March 2009

Michael Pierce, University of Texas at Austin

1. Introduction

The DTMS Framework consists of model and control libraries, fluid property calculators, and various solver routines that may be utilized to simulate physical systems. In order to be as utilitarian as possible, the framework was not written for any particular user interface; it was designed so that it could be integrated into a wide range of development environments with minimal effort. With that in mind, no interface is provided for the DTMS Framework.

Simulations are constructed and executed by creating a C++ main file and linking it to the preexisting model, control, fluid, and solver routines. Details of these routines are documented elsewhere. This tutorial is designed to walk through the steps required to create a simulation by building around a specific example that will be introduced in the next section. The main driver file that is compiled into a working executable for this simulation will be constructed one step at a time throughout each of the next few sections. At the end of each section, the code that was discussed within that section will be added to the main file, and the complete working code will be provided in the last section of this document.

2. Example

To illustrate how to construct and run a simulation within the DTMS Framework, the next few sections will walk through the steps required to build up an example system. A simplification of a water-to-water A/C plant within the starboard fresh water chilling loop of the US Navy DDG-51 destroyer will be used for this example. A more complete simulation of the starboard chilling system within the DTMS Framework is presented in the December 2007 MS Thesis by Patrick Paullus titled "Creation of a Modeling and Simulation Environment for Thermal Management of an All-Electric Ship" and in the December 2008 MS Thesis by Patrick Hewlett titled "Implementation of an In-House Framework for Dynamic Assessment of Thermal Load Management Strategies Aboard Navy Surface Ships". The layout of a highly simplified A/C system used for this tutorial is shown below in Figure 1.

The DTMS Framework is founded on the principles of Bond Graph theory wherein the primary simulation method is based on the concept of a resistance network which allows a complicated system to be represented as a Kirchoff network of resistances. There are two main classifications of models that pertain to this type of system: effort models and flow models. The terms "effort" and "flow" are used as generic, domain-independent properties whose product is the power transmitted between connected models. These have been referred to as the "across" variable and the "through" variable in other contexts. In the DTMS Framework, a flow model is one which is connected to a single inlet effort model and a single outlet effort model. By using the values of these efforts, the flow model is able to calculate the value of the flow variable that occurs between the two effort models. An effort model may be connected to any number of flow models and is responsible for calculating the effort that would be required in order for the net flow at that model to be zero. While it is not necessary for a typical user to fully understand this simulation strategy, it is important to recognize that neither two flow models nor two effort models can be directly connected. A flow model must always connect to an effort model and an effort model must always connect to a flow model.

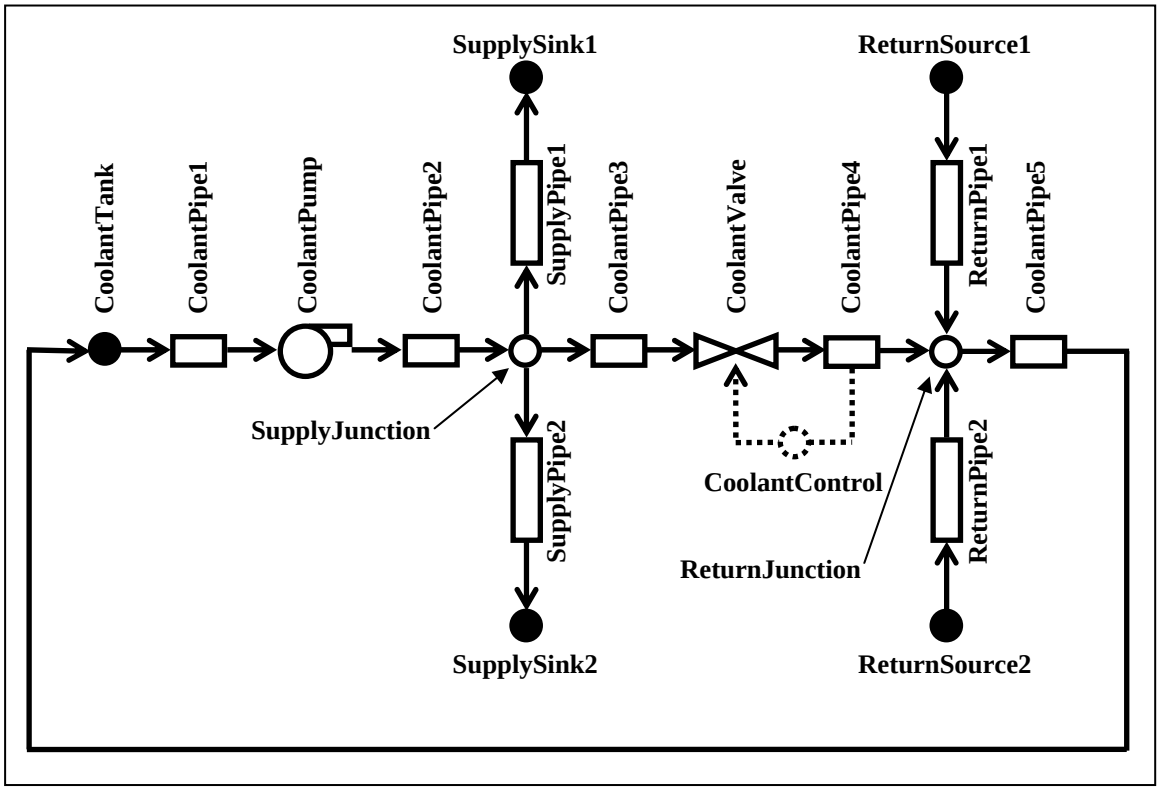


Figure 1: Mockup of simplified A/C plant

Furthermore, every system created using the DTMS Framework must be bounded by efforts that are known independently throughout the simulation. This requirement becomes trickier to recognize and satisfy as the complexity of the simulation increases.

For example, a closed loop has no obvious boundaries, but one of the efforts must be determined independently or else it will be impossible to uniquely determine the solution to the system.

Once these restrictions are understood, the example A/C plant can begin to be constructed within the DTMS Framework. In a thermal-fluid system, the variable used for the "effort" is the fluid pressure and the variable used for the "flow" is the mass flow rate of the fluid. The simplified A/C plant system consists primarily of a coolant loop, and thus there must be a pressure that is known independently within this loop in order for the system to be solved. Physically, the most logical place for this to occur is within the `CoolantTank` model, where the pressure could be calculated from the height of the fluid within the tank, or it could be maintained at a constant value by a leveling system. Choice of this model as the source of the known pressure is a decision that must be made by the modeler based on the known physics of the system.

For the sake of this tutorial, the `CoolantTank` model will be simulated using the DTMS model ThermalReservoir, which holds the supply enthalpy of the fluid constant throughout the simulation. Realistically, this model would be more accurately represented by a chiller model which employs vapor compression refrigeration to maintain chilled water enthalpy at a constant value.

The Pipe model `CoolantPipe1` connects the `CoolantTank` to the `CoolantPump`. The CentrifugalPump model `CoolantPump` then provides the flow through the A/C loop, which forces the coolant water through the rest of the system and back around to the `CoolantTank` model. The `CoolantPipe2` Pipe model carries the fluid from `CoolantPump` to the pipe junction model `SupplyJunction`, represented as a ThermalFluidEffortModel. Pipe models `CoolantPipe3` and `CoolantPipe4` carry the fluid through the control valve of the system and into the pipe junction `ReturnJunction`, again represented as a ThermalFluidEffortModel. Finally, `CoolantPipe5` returns the coolant fluid back to the `CoolantTank` to complete the coolant fluid loop.

The two independent ThermalReservoir models `ReturnSource1` and `ReturnSource2` provide the external fluid inputs to the system. These would represent a simplification of an external system which has been cooled by transferring its heat to the coolant fluid. Pipe models `ReturnPipe1` and `ReturnPipe2` bring the hot fluid back to the A/C plant, while `SupplyPipe1` and `SupplyPipe2` send the cooled fluid out to the external system which is represented by the ThermalReservoir models `SupplySink1` and `SupplySink2`.

The system is controlled by the use of the ControlValve model `CoolantValve` and the PID controller `CoolantControl`. The `CoolantControl` model monitors the exit enthalpy from `CoolantValve` and adjusts the valve position (and thus the mass flow rate) of `CoolantValve` in order to hold the exit enthalpy of `CoolantValve` at a constant value.

3. Basic C++ Structure

Now that the physics of the example model have been established, the next step is to begin writing code that will create the model within the simulation framework. First, the necessary structure of the C++ main file must be put in place. The primary component that must be included for the DTMS Framework is the following statement:

```
#include "DTMSFramework.h"
```

This file contains include statements for all of the models, controls, fluids, solvers, and other components that would be used in the construction of a simulation.

The remainder of the main file setup is discussed in following paragraphs. No other statements need to be made in order to utilize the DTMS Framework. The parameters for command-line arguments can be included, but are not required. These will not be employed in this example. The basic structure of a C++ main file is as follows:

```
#include "DTMSFramework.h"

int main()
{
      //The simulation code will be added here in later sections

      return 0;
}
```

4. Models

The next step is to create and parameterize the models that will be used in the simulation. For the specific example specified in Section 2 of this document, 18 different models will be created to simulate all physical components in the system:

- 5 ThermalReservoirs: `coolantTank`, `supplySink1`, `supplySink2`, `returnSource1`, `returnSource2`
- 9 Pipes: `coolantPipe1`, `coolantPipe2`, `coolantPipe3`, `coolantPipe4`, `coolantPipe5`, `supplyPipe1`, `supplyPipe2`, `returnPipe1`, `returnPipe2`
- 2 ThermalFluidEffortModels: `supplyJunction`, `returnJunction`
- 1 CentrifugalPump: `coolantPump`
- 1 ControlValve: `coolantValve`

First, the independent ThermalReservoir `coolantTank` is created and parameterized:

```
ThermalReservoir coolantTank("CoolantTank");
```

Notice that the constructor for the ThermalReservoir includes a character string parameter where in this case `"CoolantTank"` is provided. This parameter may be provided for any type of model created, i.e., not just ThermalReservoir models, and it represents the specific name that will be used to identify this particular model. This naming is optional and, if not provided, the name will default to `"NoName"`. These names are not required to be unique as they will only be used to identify each model when the data is output to a file.

Next we must specify that this particular model is the independent model that will be used to calculate the values for all other models in the system, which is simply done by calling the `setDependent` function:

```
coolantTank.setDependent(false);
```

The valid arguments to this function are the Boolean values `true`, which indicates that the model is dependent, and `false`, which indicates that the model is independent. Unless otherwise stated, this value defaults to `true` for every type of model within the DTMS Framework. However, since we want this model to be independent, we must provide `false` via the `setDependent` function.

The next step is to parameterize this model with known physical data. Each type of model has various parameters that are required in order for the model to work properly within a simulation. Since `CoolantTank` is the independent effort model of the system, it must provide a known (in this case constant) pressure. Similarly, the defining quality of a ThermalReservoir model is that it maintains a constant enthalpy, which may be set by the user prior to the start of the simulation. The following table contains the data that will be used to parameterize `coolantTank`:

Table 1: Physical parameters for `coolantTank`

Parameter	Value	Units
Pressure	2.0	atm
Enthalpy	28,146	J/kg

The DTMS Framework assumes certain default units when supplying data to the models, and these same units will be used when the data is output at the end of the simulation. The following table provides the default units used in DTMS:

Table 2: Default units

Parameter	Units
Length	m
Pressure	Pa
Flow Rate	kg/s
Temperature	K

Mass	kg
Density	kg/m^3
Enthalpy	kJ/kg
Entropy	kJ/kg-K
Dynamic Viscosity	Pa-s
Power	kW
Heat Rate	kW
Heat Capacity	kJ/kg-K
Thermal Conductivity	W/m-K

Since the previous units of atm and J/kg are not the default units for pressure and enthalpy, respectively, we must convert them to Pa and kJ/kg before using them in the code:

Table 3: Physical parameters for `coolantTank` in standard units

Parameter	Value	Units
Pressure	202,650	Pa
Enthalpy	28.146	kJ/kg

So we are now ready to add the necessary code that will apply these parameter values to the `coolantTank` model:

```
coolantTank.setPressure(202650);
coolantTank.set(ENTHALPY, 28.146);
```

There are two methods for providing data values to a particular model. In this case the pressure is set using an explicit set function `setPressure`. Individual models will provide the explicit set functions that they can use, and thus these will vary depending on the type of model. The most common include `setEffort` for ResistiveNetworkEffortModels, `setFlow` for ResistiveNetworkFlowModels, `setEnthalpy` for ThermalFluidModels, `setPressure` for ThermalFluidEffortModels, and `setMassFlowRate` for ThermalFluidFlowModels.

The second method for providing data is through the use of the generic `set` function, as is used to set the enthalpy for the `coolantTank` model. The first parameter to this function is an enumeration object that tells the model which data parameter is being set, and the second parameter is the data value to be used for the data parameter. The possible values for the first parameter are found throughout the various model files. This list is continually being updated as new models and new parameters are introduced. A few of these values include: `EFFORT`, `FLOW`, `ENTHALPY`, `TEMPERATURE`, `SPEED`, `DIAMETER`, `LENGTH`, `VALVE_POSITION`, `POWER`, `EFFICIENCY`, and `HEAT_LOAD`. Obviously, every model will not make use every one of these values, so a model will simply ignore the `set` function call if it does not have a need for a particular parameter.

At this point, `coolantTank` has been properly created and parameterized, so we will move on to the other models in the simulation. The next model we will focus on is the CentrifugalPump `coolantPump`, giving it the name `"CoolantPump"`:

```
CentrifugalPump coolantPump("CoolantPump");
```

The CentrifugalPump model requires several design parameters in order for it to properly calculate its state during a simulation. These include the known mass flow rate and required pressure difference at the design point, along with a maximum pressure difference achievable by the pump. Values that will be used in this simulation are provided in the table below:

Table 4: `coolantPump` design parameters

Parameter	Value	Units
Design mass flow rate	55.6	kg/s
Design pressure difference	523,265	Pa
Maximum pressure difference	777,422.9	Pa

The code needed to apply these values to the model simply requires calling the appropriate explicit set functions that are provided by the CentrifugalPump class:

```
coolantPump.setDesignFlowRate(55.6);
coolantPump.setDesignPressureDifference(523265);
coolantPump.setMaximumPressureDifference(777422.9);
```

Next, we create the ControlValve model `coolantValve` in the same manner as before:

```
ControlValve coolantValve("CoolantValve");
```

The ControlValve model operates by calculating the maximum flow rate possible when the valve is fully open based on the pressure difference between the upstream and downstream effort models. It then computes the actual flow rate using this maximum flow rate and the particular valve position. The equation used to calculate this flow is:

$$\dot{m} = \frac{VP}{100} \cdot C \cdot \left(\Delta p + H \right) \tag{1}$$

where $\dot{m}$ is the mass flow rate (kg/s), VP is the valve position in terms of percent open (%), C is the flow conductance (kg/s-Pa or m-s), Δp is the pressure difference between the upstream and downstream effort models (Pa), and H is the head due to elevation differences (Pa). The pressure difference Δp is calculated during the simulation, and the valve position VP will be automatically adjusted by the control system constructed below. Thus the only two values that must be supplied to the ControlValve model are the conductance C and the head H.

For the ControlValve `coolantValve` in our simulation, the following values are used:

Table 5: `coolantValve` design parameters

Parameter	Value	Units
Conductance when fully open	0.01	m-s
Head pressure	0.00	Pa

As before, the code to set these parameters simply involves calling the proper functions:

```
coolantValve.setOpenConductance(0.01);
coolantValve.setHeadPressure(0.0);
```

To connect our coolant loop with the hot flow returning from the external source, we need a ThermalFluidEffortModel at the junction, which is called `returnJunction`. Similarly, we need another ThermalFluidEffortModel in order to connect our coolant loop with the cold flow to the external source, which is called `supplyJunction`:

```
ThermalFluidEffortModel supplyJunction("SupplyJunction");
ThermalFluidEffortModel returnJunction("ReturnJunction");
```

These ThermalFluidEffortModels do not require any parameterization values.

A piping system is needed between the `coolantPump` and the `coolantTank`, and from the `coolantPump` to the `supplyJunction`. These are called `coolantPipe1` and `coolantPipe2`.

```
Pipe coolantPipe1("CoolantPipe1");
Pipe coolantPipe2("CoolantPipe2");
```

Also, a piping system is needed that will complete the coolant loop by connecting the `returnJunction` to the independent effort model `coolantTank`. This is called `coolantPipe5`:

```
Pipe coolantPipe5("CoolantPipe5");
```

In DTMS, Pipe models are parameterized by supplying certain geometric data about their physical structure. These required parameters include the physical length, effective length, hydraulic diameter, cross-sectional area, and head length. In this simple example, the piping system is assumed have no bends, fittings, contractions, or expansions; thus the effective length is the same as the physical length. Furthermore, it is assumed that the pipes are circular and that the hydraulic diameter is simply the physical diameter of the pipe. The values that will be used to parameterize the `coolantPipe1`, `coolantPipe2`, and `coolantPipe5` models are provided below:

Table 6: `coolantPipe1`, `coolantPipe2`, and `coolantPipe5` parameters

Parameter	Value	Units
Physical length	1.0	m
Effective length	1.0	m
Hydraulic diameter	6.0	in
Cross-sectional area	28.27	in^2
Head length	0.0	m

As before, the default unit for length is meters, and thus the values used for the hydraulic diameter and the cross-sectional area must be converted before we can proceed:

Table 7: `coolantPipe1`, `coolantPipe2`, and `coolantPipe5` parameters in standard units

Parameter	Value	Units
Physical length	1.0	m
Effective length	1.0	m
Hydraulic diameter	0.1524	m
Cross-sectional area	0.01824	m^2
Head length	0.0	m

We now apply these parameters to the Pipe models:

```
coolantPipe1.setEffectiveLength(1.0);
coolantPipe1.setPhysicalLength(1.0);
coolantPipe1.setHydraulicDiameter(0.1524);
coolantPipe1.setCrossSectionalArea(0.01824);
coolantPipe1.setHeadLength(0.0);

coolantPipe2.setEffectiveLength(1.0);
coolantPipe2.setPhysicalLength(1.0);
coolantPipe2.setHydraulicDiameter(0.1524);
coolantPipe2.setCrossSectionalArea(0.01824);
coolantPipe2.setHeadLength(0.0);

coolantPipe5.setEffectiveLength(1.0);
coolantPipe5.setPhysicalLength(1.0);
coolantPipe5.setHydraulicDiameter(0.1524);
coolantPipe5.setCrossSectionalArea(0.01824);
coolantPipe5.setHeadLength(0.0);
```

Piping systems are also needed to connect the `supplyJunction` to the `coolantValve` and the `coolantValve` to the `returnJunction`. These are called `coolantPipe3` and `coolantPipe4`. The flow in this portion of the system will be some fraction of the total flow, thus their diameter and cross-sectional area will be lower than that in the previous pipe models:

Table 8: `coolantPipe3` and `coolantPipe4` physical parameters

Parameter	Value	Units
Physical length	1.0	m
Effective length	1.0	m
Hydraulic diameter	3.0	in
Cross-sectional area	7.068	in^2
Head length	0.0	m

Again, the values used for the hydraulic diameter and the cross-sectional area must be converted to meters and square meters before we can proceed:

Table 9: `coolantPipe3` and `coolantPipe4` parameters in standard units

Parameter	Value	Units
Physical length	1.0	m
Effective length	1.0	m
Hydraulic diameter	0.0762	m
Cross-sectional area	0.00456	m^2
Head length	0.0	m

Again we apply these parameters to the Pipe models:

```
Pipe coolantPipe3("CoolantPipe3");
coolantPipe3.setEffectiveLength(1.0);
coolantPipe3.setPhysicalLength(1.0);
coolantPipe3.setHydraulicDiameter(0.0762);
coolantPipe3.setCrossSectionalArea(0.00456);
coolantPipe3.setHeadLength(0.0);

Pipe coolantPipe4("CoolantPipe4");
coolantPipe4.setEffectiveLength(1.0);
coolantPipe4.setPhysicalLength(1.0);
coolantPipe4.setHydraulicDiameter(0.0762);
coolantPipe4.setCrossSectionalArea(0.00456);
coolantPipe4.setHeadLength(0.0);
```

One final parameter that must be supplied to the `coolantPipe4` is a thermal load. Any type of ThermalFluidModel in the DTMS Framework, which includes every type of model used so far in this simulation, can have external heating applied to it. In this case, the extra heat corresponds to the waste heat dumped into the atmosphere by the A/C plant. This will simply be set to the constant value of 8.23025 kW:

```
coolantPipe4.setHeatInput(8.23025);
```

At this point, all of the models required for the coolant loop of the A/C plant have been created and properly parameterized. The remaining models required are for the branching supply and return flows. For the simplified A/C plant being modeled here, we will simply set these to be constant pressure sources and sinks with piping systems connecting them to the coolant loop. In a complete simulation of a chilling plant, such as those found in the earlier referenced work of Paullus and Hewlett (Paragraph 2), these would connect to an array of thermal loads, and their pressures and flow rates would be determined by other features of the system.

First, we must create independent effort models that will represent the pressure sources and sinks. The parameters for these models are presented below:

Table 10: Inlet and outlet effort model parameters:

	Parameter	Value	Units
SupplySink1	Pressure	820,995	Pa
	Enthalpy	28.146	kJ/kg
SupplySink2	Pressure	821,385	Pa
	Enthalpy	28.146	kJ/kg
ReturnSource1	Pressure	203,303	Pa
	Enthalpy	43.8783	kJ/kg
ReturnSource2	Pressure	202,913	Pa
	Enthalpy	43.8339	kJ/kg

We will treat these as simple ThermalReservoir models set up as for the `coolantTank`:

```
ThermalReservoir supplySink1("SupplySink1");
supplySink1.setDependent(false);
supplySink1.setPressure(820995);
supplySink1.setEnthalpy(28.146);

ThermalReservoir supplySink2("SupplySink2");
supplySink2.setDependent(false);
supplySink2.setPressure(821385);
supplySink2.setEnthalpy(28.146);

ThermalReservoir returnSource1("ReturnSource1");
returnSource1.setDependent(false);
returnSource1.setPressure(203303);
returnSource1.setEnthalpy(43.8783);

ThermalReservoir returnSource2("ReturnSource2");
returnSource2.setDependent(false);
returnSource2.setPressure(202913);
returnSource2.setEnthalpy(43.8339);
```

Next we must create the piping systems that will connect the supply and return lines to the coolant loop. For the purpose of this exercise, these are Pipe models with the following properties:

Table 11: Supply and return pipe model parameters:

Parameter	Value	Units
Physical length	10.0	m
Effective length	10.0	m
Hydraulic diameter	0.1524	m
Cross-sectional area	0.01824	m^2
Head length	0.0	m

We thus create and parameterize the four Pipe models as follows:

```
Pipe supplyPipe1("SupplyPipe1");
supplyPipe1.setEffectiveLength(10.0);
supplyPipe1.setPhysicalLength(10.0);
supplyPipe1.setHydraulicDiameter(0.1524);
supplyPipe1.setCrossSectionalArea(0.01824);
supplyPipe1.setHeadLength(0.0);

Pipe supplyPipe2("SupplyPipe2");
supplyPipe2.setEffectiveLength(10.0);
supplyPipe2.setPhysicalLength(10.0);
supplyPipe2.setHydraulicDiameter(0.1524);
supplyPipe2.setCrossSectionalArea(0.01824);
supplyPipe2.setHeadLength(0.0);

Pipe returnPipe1("ReturnPipe1");
returnPipe1.setEffectiveLength(10.0);
returnPipe1.setPhysicalLength(10.0);
returnPipe1.setHydraulicDiameter(0.1524);
returnPipe1.setCrossSectionalArea(0.01824);
returnPipe1.setHeadLength(0.0);

Pipe returnPipe2("ReturnPipe2");
returnPipe2.setEffectiveLength(10.0);
returnPipe2.setPhysicalLength(10.0);
returnPipe2.setHydraulicDiameter(0.1524);
returnPipe2.setCrossSectionalArea(0.01824);
returnPipe2.setHeadLength(0.0);
```

We have now created all of the models that appear in our simulation of the simplified A/C plant. In the next section we will create several additional models required to complete this DTMS simulation. In summary, we present the main driver file for this simulation as it now stands along with the code to create and parameterize the models:

```
#include "DTMSFramework.h"

int main()
{
      //Independent coolant tank model
      ThermalReservoir coolantTank("CoolantTank");
      coolantTank.setDependent(false);
      coolantTank.setPressure(202650);
      coolantTank.set(ENTHALPY, 28.146);

      //Centrifugal pump model
      CentrifugalPump coolantPump("CoolantPump");
      coolantPump.setDesignFlowRate(55.6);
      coolantPump.setDesignPressureDifference(523265);
      coolantPump.setMaximumPressureDifference(777422.9);

      //Control valve model
      ControlValve coolantValve("CoolantValve");
      coolantValve.setOpenConductance(0.01);
      coolantValve.setHeadPressure(0.0);

      //Supply and return junction models
      ThermalFluidEffortModel
                        supplyJunction("SupplyJunction");
      ThermalFluidEffortModel
                        returnJunction("ReturnJunction");

      //Coolant piping models
      Pipe coolantPipe1("CoolantPipe1");
      coolantPipe1.setEffectiveLength(1.0);
      coolantPipe1.setPhysicalLength(1.0);
      coolantPipe1.setHydraulicDiameter(0.1524);
      coolantPipe1.setCrossSectionalArea(0.01824);
      coolantPipe1.setHeadLength(0.0);

      Pipe coolantPipe2("CoolantPipe2");
      coolantPipe2.setEffectiveLength(1.0);
      coolantPipe2.setPhysicalLength(1.0);
      coolantPipe2.setHydraulicDiameter(0.1524);
      coolantPipe2.setCrossSectionalArea(0.01824);
      coolantPipe2.setHeadLength(0.0);

      Pipe coolantPipe3("CoolantPipe3");
      coolantPipe3.setEffectiveLength(1.0);
      coolantPipe3.setPhysicalLength(1.0);
      coolantPipe3.setHydraulicDiameter(0.0762);
      coolantPipe3.setCrossSectionalArea(0.00456);
      coolantPipe3.setHeadLength(0.0);

      Pipe coolantPipe4("CoolantPipe4");
      coolantPipe4.setEffectiveLength(1.0);
      coolantPipe4.setPhysicalLength(1.0);
```

```
coolantPipe4.setHydraulicDiameter(0.0762);
coolantPipe4.setCrossSectionalArea(0.00456);
coolantPipe4.setHeadLength(0.0);
coolantPipe4.setHeatInput(8.23025);

Pipe coolantPipe5("CoolantPipe5");
coolantPipe5.setEffectiveLength(1.0);
coolantPipe5.setPhysicalLength(1.0);
coolantPipe5.setHydraulicDiameter(0.1524);
coolantPipe5.setCrossSectionalArea(0.01824);
coolantPipe5.setHeadLength(0.0);

//Supply sink models
ThermalReservoir supplySink1("SupplySink1");
supplySink1.setDependent(false);
supplySink1.setPressure(820995);
supplySink1.setEnthalpy(28.146);

ThermalReservoir supplySink2("SupplySink2");
supplySink2.setDependent(false);
supplySink2.setPressure(821385);
supplySink2.setEnthalpy(28.146);

//Return source models
ThermalReservoir returnSource1("ReturnSource1");
returnSource1.setDependent(false);
returnSource1.setPressure(203303);
returnSource1.setEnthalpy(43.8783);

ThermalReservoir returnSource2("ReturnSource2");
returnSource2.setDependent(false);
returnSource2.setPressure(202913);
returnSource2.setEnthalpy(43.8339);

//Supply piping models
Pipe supplyPipe1("SupplyPipe1");
supplyPipe1.setEffectiveLength(10.0);
supplyPipe1.setPhysicalLength(10.0);
supplyPipe1.setHydraulicDiameter(0.1524);
supplyPipe1.setCrossSectionalArea(0.01824);
supplyPipe1.setHeadLength(0.0);

Pipe supplyPipe2("SupplyPipe2");
supplyPipe2.setEffectiveLength(10.0);
supplyPipe2.setPhysicalLength(10.0);
supplyPipe2.setHydraulicDiameter(0.1524);
supplyPipe2.setCrossSectionalArea(0.01824);
supplyPipe2.setHeadLength(0.0);

//Return piping models
Pipe returnPipe1("ReturnPipe1");
returnPipe1.setEffectiveLength(10.0);
```

```
	returnPipe1.setPhysicalLength(10.0);
	returnPipe1.setHydraulicDiameter(0.1524);
	returnPipe1.setCrossSectionalArea(0.01824);
	returnPipe1.setHeadLength(0.0);

	Pipe returnPipe2("ReturnPipe2");
	returnPipe2.setEffectiveLength(10.0);
	returnPipe2.setPhysicalLength(10.0);
	returnPipe2.setHydraulicDiameter(0.1524);
	returnPipe2.setCrossSectionalArea(0.01824);
	returnPipe2.setHeadLength(0.0);

	//The code for connections, fluids, controls,
	//solvers, and simulation execution will be added
	//here in later sections

	return 0;
}
```

5. Connections

The next step in building our simulation is to connect together the models we have just created, which is achieved by using several member functions. Recalling the initial discussion from Section 2, flow models can only be connected to a single inlet effort model and a single outlet effort model. The ResistiveNetworkFlowModel class and classes that derive from it contain the functions to perform these connections: `setInletEffortModel` and `setOutletEffortModel`. Similarly, effort models can only be connected to flow models, but they may connect to multiple inlet and outlet flow models. The ResistiveNetworkEffortModel class contains the following functions to perform these connections: `addInletFlowModel` and `addOutletFlowModel`.

It is necessary to recognize that there is a natural direction that is assigned when these functions are used: an "inlet" model is treated as "upstream" of the current model, while an "outlet" model is treated as "downstream" of the current model. These directions are merely used as a reference, and do not necessarily correspond to the intended direction of the flow. However, the direction convention must be consistent throughout the simulation. For example: let us assume that effort model `A` is connected to flow model `B`, which is then connected to effort model `C`. If `A` were chosen as the most upstream model, then `A` must be connected upstream of `B` (`A.addOutletFlowModel(B)` or `B.setInletEffortModel(A)`) and `B` must be connected upstream of `C` (`B.setOutletEffortModel(C)` or `C.addInletFlowModel(B)`). It would be incorrect to connect `C` upstream of `B` (i.e., `B.setInletEffortModel(C)` or `C.addOutletFlowModel(B)`). Similarly, `C` can be selected as the most upstream model, in which case `B` must be connected upstream of `A`. The choice of `A` or `C` as the most upstream model is irrelevant, but the proper order of the models must be maintained.

Also, the choice of function is irrelevant; each will perform the same action. While not incorrect, it is unnecessary to use both connection functions for the same two models.

In our simulation of the A/C plant, there are three main connected sections: the coolant loop, the return flows, and the supply flows. For the coolant loop, we will select the independent effort model `coolantTank` as the most upstream model and connect the rest of the models, which are all downstream, accordingly. However, we must ensure that effort models are only connected to flow models and flow models only connected to effort models. The following table presents a partial listing of the flow and effort models in the DTMS Framework:

Table 12: Flow models and effort models

Flow Models	Effort Models
`AHUDuctModel`	`CombustionChamber`
`AxialCompressor`	`CondenserOutletEffortModel`
`CentrifugalCompressor`	`EvaporatorOutletEffortModel`
`CentrifugalPump`	`TankModel`
`ControlValve`	`ThermalFluidEffortModel`
`ExpansionValve`	`ThermalReservior`
`GasTurbine`	
`LinearSpring`	
`Mass`	
`Pipe`	
`ShellModel`	
`TPShellModel`	

The connections that are needed to complete the A/C plant coolant loop are:

`coolantTank`	→	`coolantPipe1`	(effort to flow)
`coolantPipe1`	→	`coolantPump`	(flow to flow)
`coolantPump`	→	`coolantPipe2`	(flow to flow)
`coolantPipe2`	→	`supplyJunction`	(flow to effort)
`supplyJunction`	→	`coolantPipe3`	(effort to flow)
`coolantPipe3`	→	`coolantValve`	(flow to flow)
`coolantValve`	→	`coolantPipe4`	(flow to flow)
`coolantPipe4`	→	`returnJunction`	(flow to effort)
`returnJunction`	→	`coolantPipe5`	(effort to flow)
`coolantPipe5`	→	`coolantTank`	(flow to effort)

Thus, there are four situations where two flow models are connected together: `coolantPipe1` to `coolantPump`, `coolantPump` to `coolantPipe2`, `coolantPipe3` to `coolantValve`, and `coolantValve` to `coolantPipe4`. Since these models cannot be directly connected in the DTMS Framework, we must insert additional effort models into the simulation. For this task, we use objects of the ThermalFluidEffortModel class. This class does not add any additional physics to the system; it simply allows thermal/fluid properties to be passed directly from one flow model to the next. We require one of these

objects for each of the four flow-to-flow connections, and they will be referred to simply as pressure nodes, since they do not represent any physical component:

```
ThermalFluidEffortModel
                    pressureNode1("PressureNode1");
ThermalFluidEffortModel
                    pressureNode2("PressureNode2");
ThermalFluidEffortModel
                    pressureNode3("PressureNode3");
ThermalFluidEffortModel
                    pressureNode4("PressureNode4");
```

When the simulation is executed, the complexity of the system is measured by the number of dependent effort models within the system. Adding the four extra ThermalFluidEffortModels to the system will increase the complexity of our simulation, and thus increase the amount of time required to return results. For our simple simulation, this is of little consequence and can be done without a significant impact on performance. However, on a larger scale, if there were a system involving many of these A/C plants, it could be detrimental to the runtime of the simulation if extra effort models are introduced. Therefore, this solution should only be utilized in relatively small simulations.

When larger simulations are constructed, it may be beneficial to connect several flow models in series, allowing them to be treated as a single flow model and thus eliminating the need for additional effort models. In order for this solution to work, the two flow models must utilize the same basic flow equation; for example, a linear flow model should not be connected in series with a nonlinear flow model, since the resulting model will not be able to produce the same flow rate as the two models would individually. Within the DTMS Framework, models may be placed in series by creating a container model which has the same flow equation as the models it contains. For example, if we wish to connect several linear models in series, we would create a container from the LinearFlowModel class. In this case, the individual models can be added to the flow series by calling the `addSeriesFlowModel` method of the container model. While useful in larger simulations, this method will not be used in this simple example and thus will not be discussed further.

Now that we have added the four pressure nodes to our simulation, the following connections need to be made to complete the coolant loop:

```
coolantTank    → coolantPipe1      (effort to flow)
coolantPipe1   → pressureNode1     (flow to effort)
pressureNode1  → coolantPump       (effort to flow)
coolantPump    → pressureNode2     (flow to effort)
pressureNode2  → coolantPipe2      (effort to flow)
coolantPipe2   → supplyJunction    (flow to effort)
supplyJunction → coolantPipe3      (effort to flow)
```

```
coolantPipe3   → pressureNode3     (flow to effort)
pressureNode3  → coolantValve      (effort to flow)
coolantValve   → pressureNode4     (flow to effort)
pressureNode4  → coolantPipe4      (effort to flow)
coolantPipe4   → returnJunction    (flow to effort)
returnJunction → coolantPipe5      (effort to flow)
coolantPipe5   → coolantTank       (flow to effort)
```

We connect these models by using the `addOutletFlowModel` and `setOutletEffortModel` methods as follows:

```
coolantTank.addOutletFlowModel(coolantPipe1);
coolantPipe1.setOutletEffortModel(pressureNode1);
pressureNode1.addOutletFlowModel(coolantPump);
coolantPump.setOutletEffortModel(pressureNode2);
pressureNode2.addOutletFlowModel(coolantPipe2);
coolantPipe2.setOutletEffortModel(supplyJunction);
supplyJunction.addOutletFlowModel(coolantPipe3);
coolantPipe3.setOutletEffortModel(pressureNode3);
pressureNode3.addOutletFlowModel(coolantValve);
coolantValve.setOutletEffortModel(pressureNode4);
pressureNode4.addOutletFlowModel(coolantPipe4);
coolantPipe4.setOutletEffortModel(returnJunction);
returnJunction.addOutletFlowModel(coolantPipe5);
coolantPipe5.setOutletEffortModel(coolantTank);
```

Next, we connect together the flow models that are part of the supply system. The models in this section of the simulation do not require any additional pressure nodes. Thus, we can directly proceed to connecting the models. Specifically, the `setInletEffortModel` and `addInletFlowModel` methods are used in order to demonstrate that either set of methods may be used:

```
supplySink1.addInletFlowModel(supplyPipe1);
supplyPipe1.setInletEffortModel(supplyJunction);
supplyPipe2.setInletEffortModel(supplyJunction);
supplySink2.addInletFlowModel(supplyPipe2);
```

Similarly, we now connect the models of the return system, but we intentionally switch between the different connection methods to demonstrate the irrelevance of using any specific approach, so long as the upstream-downstream relationship is preserved:

```
returnSource1.addOutletFlowModel(returnPipe1);
returnJunction.addInletFlowModel(returnPipe1);
returnPipe2.setOutletEffortModel(returnJunction);
returnPipe2.setInletEffortModel(returnSource2);
```

All models have now been correctly connected together, so we present the additional code for the main driver file:

```
#include "DTMSFramework.h"

int main()
{
      /*
         ...Code that was provided in previous sections...
      */

      //Create the extra pressure nodes
      ThermalFluidEffortModel
                        pressureNode1("PressureNode1");
      ThermalFluidEffortModel
                        pressureNode2("PressureNode2");
      ThermalFluidEffortModel
                        pressureNode3("PressureNode3");
      ThermalFluidEffortModel
                        pressureNode4("PressureNode4");

      //Connect the models in the coolant loop
      coolantTank.addOutletFlowModel(coolantPipe1);
      coolantPipe1.setOutletEffortModel(pressureNode1);
      pressureNode1.addOutletFlowModel(coolantPump);
      coolantPump.setOutletEffortModel(pressureNode2);
      pressureNode2.addOutletFlowModel(coolantPipe2);
      coolantPipe2.setOutletEffortModel(supplyJunction);
      supplyJunction.addOutletFlowModel(coolantPipe3);
      coolantPipe3.setOutletEffortModel(pressureNode3);
      pressureNode3.addOutletFlowModel(coolantValve);
      coolantValve.setOutletEffortModel(pressureNode4);
      pressureNode4.addOutletFlowModel(coolantPipe4);
      coolantPipe4.setOutletEffortModel(returnJunction);
      returnJunction.addOutletFlowModel(coolantPipe5);
      coolantPipe5.setOutletEffortModel(coolantTank);

      //Connect the supply system models
      supplySink1.addInletFlowModel(supplyPipe1);
      supplyPipe1.setInletEffortModel(supplyJunction);
      supplyPipe2.setInletEffortModel(supplyJunction);
      supplySink2.addInletFlowModel(supplyPipe2);

      //Connect the return system models
      returnSource1.addOutletFlowModel(returnPipe1);
      returnJunction.addInletFlowModel(returnPipe1);
      returnPipe2.setOutletEffortModel(returnJunction);
      returnPipe2.setInletEffortModel(returnSource2);

      //The code for fluids, controls, solvers, and simulation
      //execution will be added here in later sections

      return 0;
}
```

6. Fluids

As stated in Section 2, the theoretical concepts of "flow" and "effort" are directly applicable in thermal/fluid components to the physical concepts of mass flow and pressure, respectively. These components involve physical fluids with thermodynamic properties that determine specific conditions that drive the fluid flow from one location (or model) to another. Actual systems require various fluids to operate: for example, the plumbing in a house involves water as a fluid while an air-conditioning system utilizes both air and a synthetic coolant or water.

At this point, three fluids have been completely implemented in the DTMS Framework: R-134, water, and air. Water and air each have multiple variations that may be employed; for example, air can be assumed to be calorically-perfect or simply ideal. To utilize one of these in a simulation, an instance of the desired fluid must first be created:

```
R134 myFluid1;
Water myFluid2;
CaloricallyPerfectWater myFluid3;
IdealAir myFluid4;
CaloricallyPerfectAir myFluid5;
```

In general, the thermodynamic state of each of these fluids is fixed by knowing the value of two independent, intensive properties. To update the fluid properties at any state, the DTMS user must call one of several `updateProps` functions. Currently there are four versions of `updateProps`, each requiring different known values:

- `updatePropsPT`: requires a known pressure and temperature
- `updatePropsPH`: requires a known pressure and enthalpy
- `updatePropsPS`: requires a known pressure and entropy
- `updateSatPropsP`: requires only a known pressure at the saturation point

Once one of these functions has been called, any of the following functions will return the value of the respective property for the fluid:

Table 10: Fluid properties and respective functions

Property	**Fluid function**
Temperature	`getTemperature()`
Pressure	`getPressure()`
Density	`getDensity()`
Enthalpy	`getEnthalpy()`
Entropy	`getEntropy()`
Specific heat at constant pressure	`getCp()`
Specific heat at constant volume	`getCv()`
Viscosity	`getViscosity()`

Quality	`getQuality()`
Saturated liquid density	`getSatLiqDens()`
Saturated vapor density	`getSatVapDens()`
Saturated liquid enthalpy	`getSatLiqEnth()`
Saturated vapor enthalpy	`getSatVapEnth()`
Saturated liquid specific heat at constant pressure	`getSatLiqCp()`
Saturated vapor specific heat at constant pressure	`getSatVapCp()`
Saturated liquid viscosity	`getSatLiqMu()`
Saturated vapor viscosity	`getSatVapMu()`

The simplified A/C plant simulation models a water-to-water heat exchanger, and thus each of our models must be set to utilize water as the working fluid. First, we create an instance of the fluid water that we will use for each model in the simulation:

```
Water workingFluid;
```

Then we initialize this fluid to some known state, which we will select to be the fixed pressure and enthalpy at the independent node `coolantTank`:

```
workingFluid.updatePropsPH(202650, 28.146);
```

Next, we must add the fluid to each thermal/fluid model created, which is done by calling the `setFluid` method for each of the models:

```
coolantTank.setFluid(workingFluid);
coolantPump.setFluid(workingFluid);
coolantValve.setFluid(workingFluid);
supplyJunction.setFluid(workingFluid);
returnJunction.setFluid(workingFluid);
coolantPipe1.setFluid(workingFluid);
coolantPipe2.setFluid(workingFluid);
coolantPipe3.setFluid(workingFluid);
coolantPipe4.setFluid(workingFluid);
coolantPipe5.setFluid(workingFluid);
pressureNode1.setFluid(workingFluid);
pressureNode2.setFluid(workingFluid);
pressureNode3.setFluid(workingFluid);
pressureNode4.setFluid(workingFluid);
supplySink1.setFluid(workingFluid);
supplySink2.setFluid(workingFluid);
returnSource1.setFluid(workingFluid);
returnSource2.setFluid(workingFluid);
supplyPipe1.setFluid(workingFluid);
supplyPipe2.setFluid(workingFluid);
returnPipe1.setFluid(workingFluid);
returnPipe2.setFluid(workingFluid);
```

We have now completed the code necessary for adding fluid support to each of the models, and therefore we present the new additions to the main driver file:

```
#include "DTMSFramework.h"

int main()
{
	/*
	   ...Code that was provided in previous sections...
	*/

	//Create and initialize the fluid water
	Water workingFluid;
	workingFluid.updatePropsPH(202650, 28.146);

	//Apply the fluid to each of the models
	coolantTank.setFluid(workingFluid);
	coolantPump.setFluid(workingFluid);
	coolantValve.setFluid(workingFluid);
	supplyJunction.setFluid(workingFluid);
	returnJunction.setFluid(workingFluid);
	coolantPipe1.setFluid(workingFluid);
	coolantPipe2.setFluid(workingFluid);
	coolantPipe3.setFluid(workingFluid);
	coolantPipe4.setFluid(workingFluid);
	coolantPipe5.setFluid(workingFluid);
	pressureNode1.setFluid(workingFluid);
	pressureNode2.setFluid(workingFluid);
	pressureNode3.setFluid(workingFluid);
	pressureNode4.setFluid(workingFluid);
	supplySink1.setFluid(workingFluid);
	supplySink2.setFluid(workingFluid);
	returnSource1.setFluid(workingFluid);
	returnSource2.setFluid(workingFluid);
	supplyPipe1.setFluid(workingFluid);
	supplyPipe2.setFluid(workingFluid);
	returnPipe1.setFluid(workingFluid);
	returnPipe2.setFluid(workingFluid);

	//The code for controls, solvers, and simulation
	//execution will be added here in later sections

	return 0;
}
```

7. Controls

In many simulations, it is necessary for certain constraints to be maintained even while dynamic events are occurring. For example, in the simulation of the A/C plant, we wish to maintain the enthalpy of the water at a constant value even if the fluid temperature in the supply line or return line changes or if the external heat load varies. To allow for these automated controls, the DTMS Framework provides the building blocks for creating a complex and meaningful control system.

This simulation will make use of a proportional-integral-derivative (PID) control structure. The PID controller is governed by the following transfer function:

$$G(s) = k_p\left(1 + \frac{1}{\tau_i \cdot s} + \tau_d \cdot s\right) \tag{2}$$

where k_p is the gain constant, τ_i is the integral time constant, and τ_d is the derivative time constant. In our simulation, we wish the PID controller to utilize the following constants:

Table 11: PID controller constants

Constant	Value
Gain Constant	-3
Integral Time Constant	1
Derivative Time Constant	20

The class for the PID controller is titled CTLPIDController, and it requires that the three transfer function constants be supplied in the constructor:

```
CTLPIDController coolantControl(-3, 1, 20);
```

In this simulation, we want the valve position of `coolantValve` to be modified in order to keep the enthalpy in `coolantPipe4` at a constant value. To tell the controller that it needs to be monitoring conditions in `coolantPipe4`, we call the `setMeter` function:

```
coolantControl.setMeter(coolantPipe4, ENTHALPY);
```

The first parameter for this function is either the model that should be monitored or a pointer to this object, while the second parameter is an enumeration object corresponding to the data member we wish to monitor. The possible values for the first parameter are the same values that were used in the generic `set` function that was discussed in Section 4.

We also need to tell the controller the desired value for the enthalpy, which in this case is 43.895 kJ/kg. This is accomplished by a call to the `setSetpoint` function:

```
coolantControl.setSetpoint(43.895);
```

Next, we must specify the parameter that the controller will be controlling, along with the maximum and minimum values that it can be set to. For our simulation, we want the valve position of `coolantValve` to be adjusted. Since the valve position is calculated in terms of percent, the maximum for this valve is 100, and we arbitrarily set the minimum to be 0.1. Each of these parameters can be set by using the `setDevice`, `setCeiling`, and `setFloor` functions, respectively:

```
coolantControl.setDevice(coolantValve, VALVE_POSITION);
coolantControl.setCeiling(100);
coolantControl.setFloor(0.1);
```

There are no dynamic elements in our simple A/C plant simulation, and thus automatic controls are somewhat meaningless for this system. However, by implementing a dynamic event, we can create a situation for the PID controller to control. Dynamic events allow the user to specify a new value for a property of the metered object connected to a control. In our simulation, we set an event that will reduce the heat load on `coolantPipe4` to 50% of the original value after 600 seconds:

```
coolantControl.setEvent(HEAT_LOAD, 0.50, 600);
```

Notice that the `setEvent` function requires three parameters. All of these modifications affect the metered device that is connected to `coolantControl`, which in this case is `coolantPipe4`. The first is an enumeration object corresponding to the data member that we wish to change. This is the same enumeration data we have used before, and thus the possible values are the same as used in the generic `set` function. The second parameter is a multiplier that will be applied to the data member specified by the first parameter. Since we wish to modify the heat load by 50% (or multiply it by 0.50), we have set the second parameter to 0.50. Finally, the third parameter is the time at which the event should occur. We want the change in the heat load to occur after 600 seconds, and thus we set the third parameter to be 600. Once this event has occurred, the heat load will stay at 50% for the duration of the simulation.

This completes the code necessary for addition of automatic controls and dynamic events to our simulation of the A/C plant, and thus we present the updated code for the main driver file:

```
#include "DTMSFramework.h"

int main()
{
      /*
         ...Code that was provided in previous sections...
      */

      //Create automatic controls
```

```
	CTLPIDController coolantControl(-3, 1, 20);
	coolantControl.setMeter(coolantPipe4, ENTHALPY);
	coolantControl.setSetpoint(43.895);
	coolantControl.setDevice(coolantValve, VALVE_POSITION);
	coolantControl.setCeiling(100);
	coolantControl.setFloor(0.1);
	coolantControl.setEvent(HEAT_LOAD, 0.50, 600);

	//The code for solvers and simulation execution will be
	//added here in later sections

	return 0;
}
```

8. Solvers

In order to properly simulate a given system, the relationship between the effort models and flow models becomes extremely important. At every point within the system, the flow rate into the effort models must equal the flow rate out of that same model; that is, flow conservation must be enforced at every effort model within the system. The mechanisms that accomplish this within the DTMS Framework are the solvers.

The solvers are responsible for determining the value of the effort at every dependent effort model within a given system. This is done by initially selecting an estimate for the efforts and using these estimates to calculate the flow rate through each flow model. Once the flows are known, the solvers check for flow conservation at every effort model. If this has not been achieved, then the derivative of the flow rate with respect to effort is calculated for each flow model, and these values are used to modify the efforts. The process is repeated until flow conservation has been achieved at every effort model. There are three types of solvers that are currently available in the DTMS Framework: a linearization solver, a Newton-Raphson solver, and a globally convergent solver. The linearization solver utilizes system linearization about a predefined operating point in order to approximate a solution to the system. This method is most effective when solving systems where the flow rate is a linear function of the effort, but this can also be used when the flow rate function is nonlinear. This solver requires an accurate known operating point for the system that must be relatively close to the current simulation conditions, and it provides the fastest method of solving the system. The Newton-Raphson solver allows for more accurate convergence without requiring the known operating point, instead requiring only an initial guess. This solver has a slower rate of convergence than the linearization solver, and it is more likely to produce an accurate result when the initial guess is reasonably close to the current simulation conditions. The globally convergent solver is the slowest of the three solver routines; however, it is designed to achieve near global convergence regardless of the initial guess provided to the system.

The choice of solvers depends solely on the simulation being performed. If the system consists mainly of simple flow models with linear flow functions, then the linearization solver is likely the most appropriate. If the system consists mostly of nonlinear flow models, then the Newton-Raphson solver is probably most appropriate. However, if the Newton-Raphson method is failing to converge, then it would be more appropriate to utilize the globally convergent solver.

In the simulation of the A/C plant, all of the flow models are derived from the ThermalFluidSquareRootModel base class, which has a nonlinear flow equation. Therefore, it would be most appropriate to use the Newton-Raphson solver for our simulation. We will start by creating a pointer to an instance of this type of solver:

```
NewtonRaphsonResistiveSolver simulationSolver;
```

Next, we must specify the error tolerance that will be used when checking the system for flow conservation. If the difference between the total flow rate into an effort model and the total flow rate out of the effort model is less than this error tolerance, then flow rate conservation is said to have been achieved for that model. If we chose to set it to a low value, then the output would be more accurate, whereas setting it to a high value would allow the simulation to execute more quickly. For this simulation, we set the error tolerance to 10^{-4}:

```
simulationSolver.setErrorTolerance(1e-4);
```

The last step required for initializing the solver is to add the necessary models to it. Since the solver is responsible for producing the solution to the flow network of the system, only the flow and effort models that are a part of the system need to be added to it. If multiple flow networks exist within a system, a separate solver can be used for each network, and a different type of solver can be used for each one if that is appropriate. For our simulation, all of the flow and effort models are part of the same flow network, and thus we only require the single solver. We now identify each of the flow and effort models to solver by using the `addModel` method:

```
simulationSolver.addModel(coolantTank);
simulationSolver.addModel(coolantPipe1);
simulationSolver.addModel(pressureNode1);
simulationSolver.addModel(coolantPump);
simulationSolver.addModel(pressureNode2);
simulationSolver.addModel(coolantPipe2);
simulationSolver.addModel(supplyJunction);
simulationSolver.addModel(coolantPipe3);
simulationSolver.addModel(pressureNode3);
simulationSolver.addModel(coolantValve);
simulationSolver.addModel(pressureNode4);
simulationSolver.addModel(coolantPipe4);
simulationSolver.addModel(returnJunction);
```

```
simulationSolver.addModel(coolantPipe5);
simulationSolver.addModel(supplySink1);
simulationSolver.addModel(supplySink2);
simulationSolver.addModel(supplyPipe1);
simulationSolver.addModel(supplyPipe2);
simulationSolver.addModel(returnSource1);
simulationSolver.addModel(returnSource2);
simulationSolver.addModel(returnPipe1);
simulationSolver.addModel(returnPipe2);
```

This is all that is required to initialize the system solver, so this new code is now added to the main driver file:

```
#include "DTMSFramework.h"

int main()
{
	/*
	   ...Code that was provided in previous sections...
	*/

	//Create and initialize simulation solver
	NewtonRaphsonResistiveSolver simulationSolver;
	simulationSolver.setErrorTolerance(1e-4);

	//Add all simulation models to the simulation solver
	simulationSolver.addModel(coolantTank);
	simulationSolver.addModel(coolantPipe1);
	simulationSolver.addModel(pressureNode1);
	simulationSolver.addModel(coolantPump);
	simulationSolver.addModel(pressureNode2);
	simulationSolver.addModel(coolantPipe2);
	simulationSolver.addModel(supplyJunction);
	simulationSolver.addModel(coolantPipe3);
	simulationSolver.addModel(pressureNode3);
	simulationSolver.addModel(coolantValve);
	simulationSolver.addModel(pressureNode4);
	simulationSolver.addModel(coolantPipe4);
	simulationSolver.addModel(returnJunction);
	simulationSolver.addModel(coolantPipe5);
	simulationSolver.addModel(supplySink1);
	simulationSolver.addModel(supplySink2);
	simulationSolver.addModel(supplyPipe1);
	simulationSolver.addModel(supplyPipe2);
	simulationSolver.addModel(returnSource1);
	simulationSolver.addModel(returnSource2);
	simulationSolver.addModel(returnPipe1);
	simulationSolver.addModel(returnPipe2);

	//The code for simulation execution will be added
	//here in the next section
```

```
	return 0;
}
```

9. Simulation Executive

The final step required to complete our simulation is to assimilate everything we have created thus far into the simulation executive. The simulation executive is responsible for managing every aspect of the simulation while it is running by initializing all of the models, controls, and solvers, calling the appropriate functions at the appropriate times, and writing desired data to a file.

The constructor for the DTMSSimulation class requires 4 parameters:

1) The first parameter is the name of the file to which the results of the simulation will be written. These results will be written in a CSV format, and each column header will consist of the name of the property concatenated with the name of the model. For example, the header for the temperature of `ReturnJunction` will be seen in the CSV file as `TReturnJunction`, whereas the pressure of `SupplyJunction` will be seen as `PSupplyJunction`.
2) The second parameter required by the DTMSSimulation constructor corresponds to the number of seconds to run the simulation. This number does not refer to the wall clock time, but instead corresponds to the internal simulation clock or simulation time.
3) The third parameter is the time step. This represents the time step that will be used in any time-dependent calculations within the models, and it also corresponds to the frequency at which the model parameters will be recalculated. For example, if the time step is set to 2 seconds with the simulation time at 300 seconds, then the parameters will be updated a total of 150 times; similarly if the time step is 0.1 seconds with a simulation time of 100 seconds, the parameters will be updated 1000 times.
4) The final parameter required by the DTMSSimulation constructor is the writing frequency. This is the same as the time step parameter, except that instead of updating the model parameters, the simulation executive will write the data to the file. If the write frequency is set to 2 with the total simulation time set to 300 and the time step as 1, then the model parameters will be updated every second for 300 seconds (or 300 times) while the data will be written to the file every 2 seconds (or 150 times).

For the simulation of the A/C plant, we will simply name the output file `"ACPlantSimulation.csv"`, and we will run the simulation for 900 seconds with a time step and write frequency of 1 second:

```
DTMSSimulation
```

```
executive("ACPlantSimulation.csv",900,1,1);
```

Next, we must add all of the models, controls, and solvers to the simulation executive, which should include every object that we have created up to this point. Each type of object requires its own `add` function:

```
executive.addModel(coolantTank);
executive.addModel(coolantPipe1);
executive.addModel(pressureNode1);
executive.addModel(coolantPump);
executive.addModel(pressureNode2);
executive.addModel(coolantPipe2);
executive.addModel(supplyJunction);
executive.addModel(coolantPipe3);
executive.addModel(pressureNode3);
executive.addModel(coolantValve);
executive.addModel(pressureNode4);
executive.addModel(coolantPipe4);
executive.addModel(returnJunction);
executive.addModel(coolantPipe5);
executive.addModel(supplySink1);
executive.addModel(supplySink2);
executive.addModel(supplyPipe1);
executive.addModel(supplyPipe2);
executive.addModel(returnSource1);
executive.addModel(returnSource2);
executive.addModel(returnPipe1);
executive.addModel(returnPipe2);
executive.addControl(coolantControl);
executive.addSolver(simulationSolver);
```

At this point, we must tell the simulation executive which models should have their data written to the output file. This is accomplished by calling the `setWriteFlag` function for each model and passing it either `true` or `false`. If the write flag is set to `true`, then the simulation executive will write the data from that model to the file; otherwise, the model will not appear in the data file. For our simulation, we allow each model to write its default data to the file, and thus we set the write flags of all physical components to `true`:

```
coolantTank.setWriteFlag(true);
coolantPipe1.setWriteFlag(true);
coolantPump.setWriteFlag(true);
coolantPipe2.setWriteFlag(true);
supplyJunction.setWriteFlag(true);
coolantPipe3.setWriteFlag(true);
coolantValve.setWriteFlag(true);
coolantPipe4.setWriteFlag(true);
returnJunction.setWriteFlag(true);
coolantPipe5.setWriteFlag(true);
supplySink1.setWriteFlag(true);
supplySink2.setWriteFlag(true);
```

```
supplyPipe1.setWriteFlag(true);
supplyPipe2.setWriteFlag(true);
returnSource1.setWriteFlag(true);
returnSource2.setWriteFlag(true);
returnPipe1.setWriteFlag(true);
returnPipe2.setWriteFlag(true);
```

Since the pressure nodes that were added to the simulation do not correspond to any physical component, we choose not to include their data in the output file, and set all of their write flags to `false`:

```
pressureNode1.setWriteFlag(false);
pressureNode2.setWriteFlag(false);
pressureNode3.setWriteFlag(false);
pressureNode4.setWriteFlag(false);
```

The final step required to construct our simulation is to actually initiate execution of the simulation by calling the runSimulation function from the simulation executive:

```
executive.runSimulation();
```

Once this function has been executed, the simulation will begin running for the given simulation time and will write the desired output data to the file designated.

This final code to create and initialize the simulation executive and to begin running the simulation is added to the main driver file:

```
#include "DTMSFramework.h"

int main()
{
      /*
         ...Code that was provided in previous sections...
      */

      //Create simulation executive
      DTMSSimulation
            executive("ACPlantSimulation.csv",900,1,1);

      //Add models, control, and solver to the simulation
      //executive
      executive.addModel(coolantTank);
      executive.addModel(coolantPipe1);
      executive.addModel(pressureNode1);
      executive.addModel(coolantPump);
      executive.addModel(pressureNode2);
      executive.addModel(coolantPipe2);
      executive.addModel(supplyJunction);
      executive.addModel(coolantPipe3);
      executive.addModel(pressureNode3);
```

```
        executive.addModel(coolantValve);
        executive.addModel(pressureNode4);
        executive.addModel(coolantPipe4);
        executive.addModel(returnJunction);
        executive.addModel(coolantPipe5);
        executive.addModel(supplySink1);
        executive.addModel(supplySink2);
        executive.addModel(supplyPipe1);
        executive.addModel(supplyPipe2);
        executive.addModel(returnSource1);
        executive.addModel(returnSource2);
        executive.addModel(returnPipe1);
        executive.addModel(returnPipe2);
        executive.addControl(coolantControl);
        executive.addSolver(simulationSolver);

        //Set write flags of physical components
        coolantTank.setWriteFlag(true);
        coolantPipe1.setWriteFlag(true);
        coolantPump.setWriteFlag(true);
        coolantPipe2.setWriteFlag(true);
        supplyJunction.setWriteFlag(true);
        coolantPipe3.setWriteFlag(true);
        coolantValve.setWriteFlag(true);
        coolantPipe4.setWriteFlag(true);
        returnJunction.setWriteFlag(true);
        coolantPipe5.setWriteFlag(true);
        supplySink1.setWriteFlag(true);
        supplySink2.setWriteFlag(true);
        supplyPipe1.setWriteFlag(true);
        supplyPipe2.setWriteFlag(true);
        returnSource1.setWriteFlag(true);
        returnSource2.setWriteFlag(true);
        returnPipe1.setWriteFlag(true);
        returnPipe2.setWriteFlag(true);

        //Set write flags of pressure nodes
        pressureNode1.setWriteFlag(false);
        pressureNode2.setWriteFlag(false);
        pressureNode3.setWriteFlag(false);
        pressureNode4.setWriteFlag(false);


        //Run simulation
        executive.runSimulation();

        return 0;
}
```

10. Complete Example

At this point, the construction of our main driver file is complete; it is presented below in its entirety:

```
#include "DTMSFramework.h"

int main()
{
	//Independent coolant tank model
	ThermalReservoir coolantTank("CoolantTank");
	coolantTank.setDependent(false);
	coolantTank.setPressure(202650);
	coolantTank.set(ENTHALPY, 28.146);

	//Centrifugal pump model
	CentrifugalPump coolantPump("CoolantPump");
	coolantPump.setDesignFlowRate(55.6);
	coolantPump.setDesignPressureDifference(523265);
	coolantPump.setMaximumPressureDifference(777422.9);

	//Control valve model
	ControlValve coolantValve("CoolantValve");
	coolantValve.setOpenConductance(0.01);
	coolantValve.setHeadPressure(0.0);

	//Supply and return junction models
	ThermalFluidEffortModel
			supplyJunction("SupplyJunction");
	ThermalFluidEffortModel
			returnJunction("ReturnJunction");

	//Coolant piping models
	Pipe coolantPipe1("CoolantPipe1");
	coolantPipe1.setEffectiveLength(1.0);
	coolantPipe1.setPhysicalLength(1.0);
	coolantPipe1.setHydraulicDiameter(0.1524);
	coolantPipe1.setCrossSectionalArea(0.01824);
	coolantPipe1.setHeadLength(0.0);

	Pipe coolantPipe2("CoolantPipe2");
	coolantPipe2.setEffectiveLength(1.0);
	coolantPipe2.setPhysicalLength(1.0);
	coolantPipe2.setHydraulicDiameter(0.1524);
	coolantPipe2.setCrossSectionalArea(0.01824);
	coolantPipe2.setHeadLength(0.0);

	Pipe coolantPipe3("CoolantPipe3");
	coolantPipe3.setEffectiveLength(1.0);
	coolantPipe3.setPhysicalLength(1.0);
	coolantPipe3.setHydraulicDiameter(0.0762);
```

```
coolantPipe3.setCrossSectionalArea(0.00456);
coolantPipe3.setHeadLength(0.0);

Pipe coolantPipe4("CoolantPipe4");
coolantPipe4.setEffectiveLength(1.0);
coolantPipe4.setPhysicalLength(1.0);
coolantPipe4.setHydraulicDiameter(0.0762);
coolantPipe4.setCrossSectionalArea(0.00456);
coolantPipe4.setHeadLength(0.0);
coolantPipe4.setHeatInput(8.23025);

Pipe coolantPipe5("CoolantPipe5");
coolantPipe5.setEffectiveLength(1.0);
coolantPipe5.setPhysicalLength(1.0);
coolantPipe5.setHydraulicDiameter(0.1524);
coolantPipe5.setCrossSectionalArea(0.01824);
coolantPipe5.setHeadLength(0.0);

//Supply sink models
ThermalReservoir supplySink1("SupplySink1");
supplySink1.setDependent(false);
supplySink1.setPressure(820995);
supplySink1.setEnthalpy(28.146);

ThermalReservoir supplySink2("SupplySink2");
supplySink2.setDependent(false);
supplySink2.setPressure(821385);
supplySink2.setEnthalpy(28.146);

//Return source models
ThermalReservoir returnSource1("ReturnSource1");
returnSource1.setDependent(false);
returnSource1.setPressure(203303);
returnSource1.setEnthalpy(43.8783);

ThermalReservoir returnSource2("ReturnSource2");
returnSource2.setDependent(false);
returnSource2.setPressure(202913);
returnSource2.setEnthalpy(43.8339);

//Supply piping models
Pipe supplyPipe1("SupplyPipe1");
supplyPipe1.setEffectiveLength(10.0);
supplyPipe1.setPhysicalLength(10.0);
supplyPipe1.setHydraulicDiameter(0.1524);
supplyPipe1.setCrossSectionalArea(0.01824);
supplyPipe1.setHeadLength(0.0);

Pipe supplyPipe2("SupplyPipe2");
supplyPipe2.setEffectiveLength(10.0);
supplyPipe2.setPhysicalLength(10.0);
supplyPipe2.setHydraulicDiameter(0.1524);
```

```
supplyPipe2.setCrossSectionalArea(0.01824);
supplyPipe2.setHeadLength(0.0);

//Return piping models
Pipe returnPipe1("ReturnPipe1");
returnPipe1.setEffectiveLength(10.0);
returnPipe1.setPhysicalLength(10.0);
returnPipe1.setHydraulicDiameter(0.1524);
returnPipe1.setCrossSectionalArea(0.01824);
returnPipe1.setHeadLength(0.0);

Pipe returnPipe2("ReturnPipe2");
returnPipe2.setEffectiveLength(10.0);
returnPipe2.setPhysicalLength(10.0);
returnPipe2.setHydraulicDiameter(0.1524);
returnPipe2.setCrossSectionalArea(0.01824);
returnPipe2.setHeadLength(0.0);

//Create the extra pressure nodes
ThermalFluidEffortModel
                  pressureNode1("PressureNode1");
ThermalFluidEffortModel
                  pressureNode2("PressureNode2");
ThermalFluidEffortModel
                  pressureNode3("PressureNode3");
ThermalFluidEffortModel
                  pressureNode4("PressureNode4");

//Connect the models in the coolant loop
coolantTank.addOutletFlowModel(coolantPipe1);
coolantPipe1.setOutletEffortModel(pressureNode1);
pressureNode1.addOutletFlowModel(coolantPump);
coolantPump.setOutletEffortModel(pressureNode2);
pressureNode2.addOutletFlowModel(coolantPipe2);
coolantPipe2.setOutletEffortModel(supplyJunction);
supplyJunction.addOutletFlowModel(coolantPipe3);
coolantPipe3.setOutletEffortModel(pressureNode3);
pressureNode3.addOutletFlowModel(coolantValve);
coolantValve.setOutletEffortModel(pressureNode4);
pressureNode4.addOutletFlowModel(coolantPipe4);
coolantPipe4.setOutletEffortModel(returnJunction);
returnJunction.addOutletFlowModel(coolantPipe5);
coolantPipe5.setOutletEffortModel(coolantTank);

//Connect the supply system models
supplySink1.addInletFlowModel(supplyPipe1);
supplyPipe1.setInletEffortModel(supplyJunction);
supplyPipe2.setInletEffortModel(supplyJunction);
supplySink2.addInletFlowModel(supplyPipe2);

//Connect the return system models
returnSource1.addOutletFlowModel(returnPipe1);
```

```
returnJunction.addInletFlowModel(returnPipe1);
returnPipe2.setOutletEffortModel(returnJunction);
returnPipe2.setInletEffortModel(returnSource2);

//Create and initialize the fluid water
Water workingFluid;
workingFluid.updatePropsPH(202650, 28.146);

//Apply the fluid to each of the models
coolantTank.setFluid(workingFluid);
coolantPump.setFluid(workingFluid);
coolantValve.setFluid(workingFluid);
supplyJunction.setFluid(workingFluid);
returnJunction.setFluid(workingFluid);
coolantPipe1.setFluid(workingFluid);
coolantPipe2.setFluid(workingFluid);
coolantPipe3.setFluid(workingFluid);
coolantPipe4.setFluid(workingFluid);
coolantPipe5.setFluid(workingFluid);
pressureNode1.setFluid(workingFluid);
pressureNode2.setFluid(workingFluid);
pressureNode3.setFluid(workingFluid);
pressureNode4.setFluid(workingFluid);
supplySink1.setFluid(workingFluid);
supplySink2.setFluid(workingFluid);
returnSource1.setFluid(workingFluid);
returnSource2.setFluid(workingFluid);
supplyPipe1.setFluid(workingFluid);
supplyPipe2.setFluid(workingFluid);
returnPipe1.setFluid(workingFluid);
returnPipe2.setFluid(workingFluid);

//Create automatic controls
CTLPIDController coolantControl(-3, 1, 20);
coolantControl.setMeter(coolantPipe4, ENTHALPY);
coolantControl.setSetpoint(43.895);
coolantControl.setDevice(coolantValve,
      VALVE_POSITION);
coolantControl.setCeiling(100);
coolantControl.setFloor(0.1);
coolantControl.setEvent(HEAT_LOAD, 0.50, 600);

//Create and initialize simulation solver
NewtonRaphsonResistiveSolver simulationSolver;
simulationSolver.setErrorTolerance(1e-4);

//Add all simulation models to the simulation solver
simulationSolver.addModel(coolantTank);
simulationSolver.addModel(coolantPipe1);
simulationSolver.addModel(pressureNode1);
simulationSolver.addModel(coolantPump);
simulationSolver.addModel(pressureNode2);
simulationSolver.addModel(coolantPipe2);
```

```
	simulationSolver.addModel(supplyJunction);
	simulationSolver.addModel(coolantPipe3);
	simulationSolver.addModel(pressureNode3);
	simulationSolver.addModel(coolantValve);
	simulationSolver.addModel(pressureNode4);
	simulationSolver.addModel(coolantPipe4);
	simulationSolver.addModel(returnJunction);
	simulationSolver.addModel(coolantPipe5);
	simulationSolver.addModel(supplySink1);
	simulationSolver.addModel(supplySink2);
	simulationSolver.addModel(supplyPipe1);
	simulationSolver.addModel(supplyPipe2);
	simulationSolver.addModel(returnSource1);
	simulationSolver.addModel(returnSource2);
	simulationSolver.addModel(returnPipe1);
	simulationSolver.addModel(returnPipe2);

	//Create simulation executive
	DTMSSimulation
		executive("ACPlantSimulation.csv",900,1,1);

	//Add models, control, and solver to the simulation
	//executive
	executive.addModel(coolantTank);
	executive.addModel(coolantPipe1);
	executive.addModel(pressureNode1);
	executive.addModel(coolantPump);
	executive.addModel(pressureNode2);
	executive.addModel(coolantPipe2);
	executive.addModel(supplyJunction);
	executive.addModel(coolantPipe3);
	executive.addModel(pressureNode3);
	executive.addModel(coolantValve);
	executive.addModel(pressureNode4);
	executive.addModel(coolantPipe4);
	executive.addModel(returnJunction);
	executive.addModel(coolantPipe5);
	executive.addModel(supplySink1);
	executive.addModel(supplySink2);
	executive.addModel(supplyPipe1);
	executive.addModel(supplyPipe2);
	executive.addModel(returnSource1);
	executive.addModel(returnSource2);
	executive.addModel(returnPipe1);
	executive.addModel(returnPipe2);
	executive.addControl(coolantControl);
	executive.addSolver(simulationSolver);

	//Set write flags of physical components
	coolantTank.setWriteFlag(true);
	coolantPipe1.setWriteFlag(true);
	coolantPump.setWriteFlag(true);
	coolantPipe2.setWriteFlag(true);
```

```
	supplyJunction.setWriteFlag(true);
	coolantPipe3.setWriteFlag(true);
	coolantValve.setWriteFlag(true);
	coolantPipe4.setWriteFlag(true);
	returnJunction.setWriteFlag(true);
	coolantPipe5.setWriteFlag(true);
	supplySink1.setWriteFlag(true);
	supplySink2.setWriteFlag(true);
	supplyPipe1.setWriteFlag(true);
	supplyPipe2.setWriteFlag(true);
	returnSource1.setWriteFlag(true);
	returnSource2.setWriteFlag(true);
	returnPipe1.setWriteFlag(true);
	returnPipe2.setWriteFlag(true);

	//Set write flags of pressure nodes
	pressureNode1.setWriteFlag(false);
	pressureNode2.setWriteFlag(false);
	pressureNode3.setWriteFlag(false);
	pressureNode4.setWriteFlag(false);

	//Run simulation
	executive.runSimulation();

	return 0;
}
```

Appendix D: DTMS Input Deck Specification

Version 1.0.2, Mar. 16, 2009

Tomasz Haupt, Mississippi State University
Michael Pierce, University of Texas at Austin

1. Syntax

Data cards

The DTMS input deck is sequence of lines referred to as "cards". The format of a card is modeled after Fortran NAMELIST statement, and looks as follows:

```
&KEYWORD variable1=3.14, variable2=.false., variable3='string',...
```

Comments

"!" symbol as the first character in the line makes the parser to ignore the contents of the line (that is, making it a comment); examples

```
!&KEYWORD variable1=5, variable2=.false.

! Another comment line
```

All characters after "/" symbol are ignored, too, e.g.,

```
&KEYWORD variable1=0.1, variable2=.false. /a comment on this line.
```

Data types

Type	Example
real	a=1.567
integer	b=5
logical	c=.false., d=.true.
text	d='all strings are CaSE sENsitiVe'
array	z=1,2,3, x=12,14.5,'abc',.true.

2. DTMS cards

There are currently 6 keywords needed to express a DTMS model: Solver, Fluid, FlowModel, EffortModel, DTMSModel, and Control. The number of keywords will change as the framework is expanded and refined, but these 6 keywords are capable of representing every possible configuration currently supported in DTMS. Soon, the specification for a type of model which encapsulates multiple EffortModels and FlowModels will be complete, which will add a 7th keyword: GroupModel.

The list of variables following the keyword in the card changes from card to card ("is context sensitive"), however each card has 'id', 'name', and 'type' variables. Some of the variables are mandatory, the other are optional. If an optional variable is missing in the card, a default value is assumed.

'id' variable provides a unique object identifier for a DTMS object instantiated as the result of processing the card. The uniqueness is required only for objects of the same type. Assigning an id for each object seems to be a nuisance when generating the cards manually. However, ids are very convenient when the cards are generated automatically from a database: each id is simply the primary key in the database. In all examples given below, the ids are small consecutive integers; no such restrictions are needed.

'name' is a name of the object instance to be displayed to the user (a label). In all examples given below we use variable names used in the DTMS tutorial because they are very descriptive. No relation between the variable names and the labels defined by the 'name' variable is implied, however.

'type' is the name of the class representing the object to be instantiated as the result of processing the card. A strict one-to-one correspondence is mandatory.

2.1 Solver

```
&SOLVER id=integer, name=string, type=string, solver-type-dependent
variables
```

The possible types of Solver include:

- GloballyConvergentResistiveSolver
- NewtonRaphsonResistiveSolver

The required and optional variables for these Solvers are presented in Section 5 of this document.

Example:

```
&SOLVER id=1, name='simulationSolver',
type='NewtonRaphsonResistiveSolver', errorTolerance=0.0001
```

2.2 Fluid

```
&FLUID id=integer, name=string, type=string, temperature=real,
pressure=real, density=real, ..., satVaporViscosity=real,…
```

All variables starting with temperature are optional, except for ‘id’, ‘name’, and ‘type’. If no values are provided for the optional variables, the default values are used.

The possible types of Fluid include:

- CaloricallyPerfectAir
- CaloricallyPerfectWater
- Fuel
- IdealAir
- R134
- Water

The required and optional variables for these Fluids are presented in Section 5.

Example:

```
&FLUID id=1, name='workingFluid1', type='r134', pressure=202650.0,
enthalpy=28.146
```

2.3 EffortModel

```
&EFFORTMODEL id=integer, name=string, type=string, dependent=logical,
output=logical, solverID=integer, model-type-dependent variables
```

The possible types of EffortModel include:

- CombustionChamber
- ThermalFluidEffortModel
- ThermalReservoir

The required and optional variables for these EffortModels are presented in Section 5 of this document.

Examples:

```
&EFFORTMODEL id=1, name='source1', type='ThermalReservoir'
dependent=.false., output=.false., solverID=1, fluidID=1,
pressure=202650.0, enthalpy=28.146
&EFFORTMODEL id=2, name='pressureNode1',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1
```

Admittedly, the names of the variables are ugly in these examples, we will need to come with better ones.

***Note**: the "definition" of the EFFORTMODEL id=1, i.e. source1 of type ThermalFluidEffortModel, not only comprises the node parameters (such as pressure and enthalpy), but also specify the working fluid (defined with id=1 in this example), solver to be applied (defined with id=1; since object workingFluid and Solver are of different type, there is nothing wrong with both having id=1), and output flag.*

2.4 FlowModel

```
&FLOWMODEL id=integer, name=string, type=string, dependent=logical,
output=logical, solverID=integer, effortIDs=integer_array, model-type-
dependent variables
```

The possible types of FlowModel include:

- AxialCompressor
- CentrifugalPump
- ControlValve
- GasTurbine
- Pipe
- ShellModel
- ThermalFluidSquareRootFlowModel

The required and optional variables for these FlowModels are presented in Section 5 of this document.

***Note:** Here it is assumed that a FlowModel is associated with a physical object with two ends, with each end represented by an effort model. The instances of the effort models on each end are identified by a pair of EffortModel ids given as an array value of the effortIDs variable. The order of these ids is significant. The nominal flow direction goes always from the first id to the other one. Perhaps it would be better to set these as two separate variables: 'upstreamEffortID' and 'downstreamEffortID'.*

Examples:

```
&FLOWMODEL id=1, name='coolantPump', type='CentrifugalPump',
dependent=.true., output=.true., solverID=1, effortIDs=2,3, fluidID=1,
designFlowRate=55.6, designPressureDifference=523265.0,
maximumPressureDifference=777422.9

&FLOWMODEL id=2, name='coolantPipe4' type='Pipe', dependent=.true.,
output=.true., solverID=1, effortIDs=6,7, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=3.0,
crossSectionalArea=0.00456, headDistance=0.0, heatInput=8.23025
```

2.5 DTMSModel

```
&DTMSModel, id=integer, name=string, type=string, dependent=logical,
output=logical, type-specific parameters
```

This model was not used in the DTMS tutorial, but it represents any model that does not fit the requirements of either a FlowModel or an EffortModel.

The possible types of DTMSModel include:

- Shaft
- ShellTubeHX
- WWHeatExchanger

The required and optional variables for these DTMSModels are presented in Section 5 of this document.

Each of these has their own requirements for how they are connected to other models, along with their own optional and required variables. In the future, 'WWHeatExchanger' and 'ShellTubeHX' will be converted into GroupModels, while Shaft may be converted into an EffortModel. This type of model may be removed at some point, but it is currently seen as a crucial part of the DTMS Framework.

Example:

```
&DTMSModel, id=1, name='loadShaft' type='Shaft', dependent=.true.,
output=.true., flowIDs=1,3, inertia=700, fixedLosses=0.0,
viscousLosses=0.0, initialSpeed=3600
```

2.6 Control

```
&CONTROL, id=integer, name=string, type=string, monitorID=integer,
monitorVariable=string, deviceID=integer, deviceVariable=string, type-
specific parameters
```

The possible types of Control include:

- CTLPIDController

The required and optional variables for these Controls are presented in Section 5 of this document.

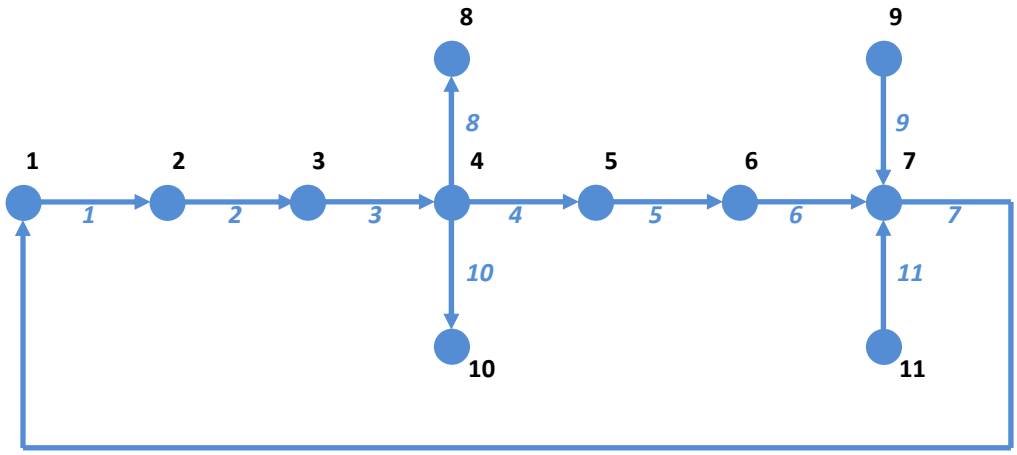
Example:

```
&CONTROL, id=1, name='coolantController' type='CTLPIDController',
monitorID=2, monitorVariable='ENTHALPY', deviceID=3,
deviceVariable='VALVE_POSITION', gainConstant=-3, ITconstant=1,
```

```
DTconstant=20, ceiling=100.0, floor=0.1, setPoint=43.895,
event='HEAT_LOAD',0.50,600
```

Admittedly, this card definition is the least mature, and similarly, this is the least mature aspect of the DTMS Framework. At this point, the 'CTLPIDController' is the only type control that is designed to be used in a simulation, and the general direction for the control system is not well defined. Further development of this aspect of the framework is currently being addressed within the Thermal Management group at the University of Texas. At this point, the general use of 'monitorID' and 'deviceID' pose a problem, since the control could be monitoring either an EffortModel, a FlowModel, or a DTMSModel, and there appears to be no way to distinguish between these types of IDs.

3. Example A/C plant



This is a simplified version of the example A/C plant given in the DTMS tutorial. In this example we tried to avoid the creation of any model containers, creating effective models, etc. This version roughly represents what we expect to extract from a CAD file with minimal intelligence built into it. The only objects that would not be found in the CAD file are the 4 pressure nodes, which represent imaginary pressure sensors and are required in order to properly connect the network in DTMS. In a long haul we expect this model to be optimized one way or another, but it seems to be a good target for early experiments.

Effort Models (black labels):

1. Coolant Tank
2. Pressure Node 1
3. Pressure Node 2
4. Supply Junction
5. Pressure Node 3
6. Pressure Node 4
7. Return Junction
8. Supply Sink 1

9. Supply Sink 2
10. Return Source 1
11. Return Source 2

Flow Models (blue labels in italics):

1. Coolant Pipe 1
2. Coolant Pump
3. Coolant Pipe 2
4. Coolant Pipe 3
5. Valve
6. Coolant Pipe 4
7. Coolant Pipe 5
8. Supply Pipe 1
9. Supply Pipe 2
10. Return Pipe 1
11. Return Pipe 2

4. Input deck for the example A/C plant

```
&SOLVER id=1, name='simulationSolver',
type='NewtonRaphsonResistiveSolver', errorTolerance=0.0001

&FLUID id=1, name='workingFluid1', type='Water', pressure=202650.0,
enthalpy=28.146

! Effort models

&EFFORTMODEL id=1, name='coolantTank', type='ThermalReservoir',
dependent=.true., output=.true., solverID=1, fluidID=1,
pressure=202650.0, enthalpy=28.146

&EFFORTMODEL id=2, name='pressureNode1',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1

&EFFORTMODEL id=3, name='pressureNode2',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1

&EFFORTMODEL id=4, name='supplyJunction',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1

&EFFORTMODEL id=5, name='pressureNode3',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1
&EFFORTMODEL id=6, name='pressureNode4',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1
```

```
&EFFORTMODEL id=7, name='returnJunction',
type='ThermalFluidEffortModel', dependent=.true., output=.true.,
solverID=1, fluidID=1

&EFFORTMODEL id=8, name='supplySource1', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=1,
pressure=203303.0, enthalpy=43.8783

&EFFORTMODEL id=9, name='supplySource2', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=1,
pressure=203303.0, enthalpy=43.8783

&EFFORTMODEL id=10, name='returnSink1', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=1,
pressure=820995.0, enthalpy=28.146

&EFFORTMODEL id=11, name='returnSink2', type='ThermalReservoir',
dependent=.false., output=.true., solverID=1, fluidID=1,
pressure=820995.0, enthalpy=28.146

! Flow models

&FLOWMODEL id=1, name=’coolantPipe1’, type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=1,2, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=2, name='coolantPump', type='CentrifugalPump',
dependent=.true., output=.true., solverID=1, effortIDs=2,3, fluidID=1,
designFlowRate=55.6, designPressureDifference=523265.0,
maximumPressureDifference=777422.9

&FLOWMODEL id=3, name=’coolantPipe2’, type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=3,4, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=4, name=’coolantPipe3’, type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=4,5, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.0762,
crossSectionalArea=0.00456, headDistance=0.0

&FLOWMODEL id=5, name='coolantValve', type='ControlValve',
dependent=.true., output=.true., solverID=1, effortIDs=5,6, fluidID=1,
openConductance=0.01

&FLOWMODEL id=6, name=’coolantPipe4’, type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=6,7, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.0762,
crossSectionalArea=0.00456, headDistance=0.0, heatInput=8.23025
```

```
&FLOWMODEL id=7, name=’coolantPipe5’, type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=7,1, fluidID=1,
physicalLength=1.0, effectiveLength=1.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=8, name='supplyPipe1', type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=4,8, fluidID=1,
physicalLength=10.0, effectiveLength=10.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=9, name='supplyPipe2', type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=4,10, fluidID=1,
physicalLength=10.0, effectiveLength=10.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=10, name='returnPipe1', type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=9,7, fluidID=1,
physicalLength=10.0, effectiveLength=10.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

&FLOWMODEL id=11, name='returnPipe2', type=’Pipe’, dependent=.true.,
output=.true., solverID=1, effortIDs=11,7, fluidID=1,
physicalLength=10.0, effectiveLength=10.0, hydraulicDiameter=0.1524,
crossSectionalArea=0.01824, headDistance=0.0

! Controls

&CONTROL, id=1, name='coolantController' type='CTLPIDController',
monitorID=6, monitorVariable=’ENTHALPY’, deviceID=5,
deviceVariable=’VALVE_POSITION’, gainConstant=-3, ITconstant=1,
DTconstant=20, ceiling=100, floor=0.1, setPoint=43.895,
event='HEAT_LOAD',0.50,600
```

5. Index of DTMS Group Names and Properties

Types of Solvers

`NewtonRaphsonResistiveSolver`

Required:

N/A

Optional:

`errorTolerance=real`

`GloballyConvergentResistiveSolver`

Required:

N/A

Optional:

`errorTolerance=real`

Types of Fluids

`IdealAir`

Required:

N/A

Optional:

```
massPercent=integer,real
referencePressure=integer,real
referenceTemperature=integer,real
referenceEnthalpy=integer,real
referenceEntropy=integer,real
relativeHumidity=real,real,real
absoluteHumidity=real,real,real
specificHumidity=real
pressure=real
temperature=real
density=real
enthalpy=real
entropy=real
cp=real
cv=real
viscosity=real
```

`CaloricallyPerfectAir`

Required:

N/A

Optional:

```
massPercent=integer,real
referencePressure=integer,real
referenceTemperature=integer,real
referenceEnthalpy=integer,real
referenceEntropy=integer,real
relativeHumidity=real,real,real
absoluteHumidity=real,real,real
specificHumidity=real
pressure=real
temperature=real
density=real
enthalpy=real
entropy=real
cp=real
cv=real
viscosity=real
```

`Fuel`

Required:

N/A

Optional:

```
lowerHeatingValue=real
massPercent=integer,real
pressure=real
temperature=real
density=real
enthalpy=real
```

```
            entropy=real
            cp=real
            cv=real
            viscosity=real
```

`R134`

Required:

N/A

Optional:

```
            pressure=real
            temperature=real
            density=real
            enthalpy=real
            entropy=real
            cp=real
            cv=real
            viscosity=real
            satLiqDens=real
            satVapDens=real
            satLiqEnth=real
            satVapEnth=real
            satLiqMu=real
            satVapMu=real
            satLiqCp=real
            satVapCp=real
            quality=real
```

`Water`

Required:

N/A

Optional:

```
            pressure=real
            temperature=real
            density=real
            enthalpy=real
            entropy=real
            cp=real
            cv=real
            viscosity=real
            satLiqDens=real
            satVapDens=real
            satLiqEnth=real
            satVapEnth=real
            satLiqMu=real
            satVapMu=real
            satLiqCp=real
            satVapCp=real
            quality=real
```

`CaloricallyPerfectWater`

Required:

N/A

Optional:

```
pressure=real
temperature=real
density=real
enthalpy=real
entropy=real
cp=real
cv=real
viscosity=real
```

Types of EffortModels

`CombustionChamber`

Required:

```
dependence=logical
output=logical
solverID=integer
fluidID=integer
```

Optional:

```
designFuelFlowIn=real
efficiency=real
enthalpy=real
pressure=real
```

`ThermalFluidEffortModel`

Required:

```
dependence=logical
output=logical
solverID=integer
fluidID=integer
```

Optional:

```
enthalpy=real
pressure=real
```

`ThermalReservoir`

Required:

```
dependence=logical
output=logical
solverID=integer
fluidID=integer
```

Optional:

```
enthalpy=real
pressure=real
```

Types of FlowModels

`AxialCompressor`

Required:

```
dependence=logical
output=logical
solverID=integer
```

effortIDs=integer_array
fluidID=integer

Optional:

designFlowRate=real
designInletPressure=real
designInletEnthalpy=real
designPower=real
designPressureRatio=real
designSpeed=real
enthalpy=real

CentrifugalPump

Required:

dependence=logical
output=logical
solverID=integer
effortIDs=integer_array
fluidID=integer

Optional:

designDensity=real
designFlowRate=real
designEfficiency=real
designPressureDifference=real
designSpeed=real
enthalpy=real
maximumPressureDifference=real

ControlValve

Required:

dependence=logical
output=logical
solverID=integer
effortIDs=integer_array
fluidID=integer

Optional:

enthalpy=real
openConductance=real
position=real

GasTurbine

Required:

dependence=logical
output=logical
solverID=integer
effortIDs=integer_array
fluidID=integer

Optional:

designFlowRate=real
designInletPressure=real
designInletEnthalpy=real

```
            designPower=real
            designPressureRatio=real
            designSpeed=real
            enthalpy=real

    Pipe
        Required:
            dependence=logical
            output=logical
            solverID=integer
            effortIDs=integer_array
            fluidID=integer
        Optional:
            crossSectionalArea=real
            effectiveLength=real
            enthalpy=real
            headDistance=real
            hydraulicDiameter=real
            perimeter=real
            physicalLength=real
            roughness=real
            thickness=real

    ShellModel
        Required:
            dependence=logical
            output=logical
            solverID=integer
            effortIDs=integer_array
            fluidID=integer
        Optional:
            diameter=real
            enthalpy=real
            length=real
            tubeLayout=integer
            wallTemperature=real

    ThermalFluidSquareRootFlowModel
        Required:
            dependence=logical
            output=logical
            solverID=integer
            effortIDs=integer_array
            fluidID=integer
        Optional:
            enthalpy=real
            flowCoefficient=real
            sourceTerm=real
```

Types of DTMSModels

```
    Shaft
```

Required:

```
dependence=logical
output=logical
flowIDs=integer_array
```

Optional:

```
fixedLosses=real
inertia=real
initialSpeed=real
turningGearSpeed=real
viscousLosses=real
```

`ShellTubeHX`

Required:

```
dependence=logical
output=logical
effortIDs=integer_array
```

Optional:

```
N/A
```

`WWHeatExchanger`

Required:

```
dependence=logical
output=logical
effortIDs=integer_array
```

Optional:

```
N/A
```

Types of Controls

`CTLPIDController`

Required:

```
monitorID=integer
monitorVariable=string
deviceID=integer
deviceVariable=string
gainConstant=real
ITConstant=real
DTConstant=real
ceiling=real
floor=real
setpoint=real
```

Optional:

```
event=string,real,real
```

Appendix E: `ResistiveNetworkGroupModel` Source Files

Below is the header file used to define the DTMS model of the `RollsRoyceMT30`, which demonstrates the usage of the `ResistiveNetworkGroupModel` class of the DTMS Framework.

```
#include "../../Base/DTMSFluid.h"
#include "../../ModelTemplates/ShaftModel.h"
#include "../Base/ResistiveNetworkGroupModel.h"
#include "../ThermalFluid/ThermalFluidEffortModel.h"
#include "AxialCompressor.h"
#include "CombustionChamber.h"
#include "ControlValve.h"
#include "GasTurbine.h"
#include "Shaft.h"

namespace DTMSFramework {

class RollsRoyceMT30 : public ResistiveNetworkGroupModel, public
      ShaftModel
{
private:
      /** Stores the class name of this model (RollsRoyceMT30) */
      static const std::string className_;

public:
      /** Creates a factory plugin for this model, which allows it to
            be created through a scripting interface. */
      static const ResistiveGroupModelFactoryPlugin<RollsRoyceMT30>
            plugin;

protected:

      AxialCompressor IPC_;
      AxialCompressor HPC_;
      CombustionChamber combustor_;
      GasTurbine HPT_;
      GasTurbine IPT_;
      GasTurbine FPT_;

      ThermalFluidEffortModel compressorNode_;
      ThermalFluidEffortModel upperTurbineNode_;
      ThermalFluidEffortModel lowerTurbineNode_;

      Shaft lowPressureShaft_;
      Shaft highPressureShaft_;
```

```
      ControlValve fuelValve_;

      DTMSFluid * inletAirFluid_;
      DTMSFluid * outletAirFluid_;
      DTMSFluid * inletFuelFluid_;

      bool isFluidPerfect_;

public:

      //---------------------------------------------------------------
      // Constructors
      //---------------------------------------------------------------

      RollsRoyceMT30();

      RollsRoyceMT30(const std::string & name);


      //---------------------------------------------------------------
      // Simulation methods
      //---------------------------------------------------------------

      virtual void setDefaults();

      virtual void initialize();


      //---------------------------------------------------------------
      // Connection methods
      //---------------------------------------------------------------

      //Effort connection methods
      virtual void setInletAirEffortModel(ThermalFluidEffortModel *
            inletAirEffortModel);

      virtual void setInletAirEffortModel(ThermalFluidEffortModel &
            inletAirEffortModel);

      virtual void setInletFuelEffortModel(ThermalFluidEffortModel *
            inletFuelEffortModel);

      virtual void setInletFuelEffortModel(ThermalFluidEffortModel &
            inletFuelEffortModel);

      virtual void setOutletAirEffortModel(ThermalFluidEffortModel *
            outletAirEffortModel);

      virtual void setOutletAirEffortModel(ThermalFluidEffortModel &
            outletAirEffortModel);

      //Shaft connection methods
      void setShaft(Shaft * shaft);
```

```
      void setShaft(Shaft & shaft);


      //-------------------------------------------------------------
      // Access methods for private data members
      //-------------------------------------------------------------

      //DTMS Model members
      virtual void setName(const std::string & name);

      virtual void setWriteFlag(bool writeFlag);

      //Fluid members
      virtual void setInletAirFluid(DTMSFluid * fluid);

      virtual void setInletAirFluid(DTMSFluid & fluid);

      virtual void setInletFuelFluid(DTMSFluid * fluid);

      virtual void setInletFuelFluid(DTMSFluid & fluid);

      virtual void setOutletAirFluid(DTMSFluid * fluid);

      virtual void setOutletAirFluid(DTMSFluid & fluid);

      virtual void setIsFluidPerfect(bool isFluidPerfect);

      //Shaft members
      virtual void setSpeed(double speed);

      virtual double getSpeed();

      virtual void setPower(double power);

      virtual double getPower();

      virtual void setInertia(double inertia);

      virtual double getInertia();


      //-------------------------------------------------------------
      // Methods for communicating data between objects
      //-------------------------------------------------------------

      virtual void set(const DTMSData & variable, double value) ;

      virtual double get(const DTMSData & variable) ;
};

}
```

Below is the implementation file used to describe the programming logic of the DTMS model of the `RollsRoyceMT30`, which further demonstrates the usage of the `ResistiveNetworkGroupModel` class of the DTMS Framework.

```
#include "RollsRoyceMT30.h"
#include "../../Base/DTMSData.h"

#include <cmath>

using namespace std;

namespace DTMSFramework {

const string RollsRoyceMT30::className_ = "RollsRoyceMT30";
const ResistiveGroupModelFactoryPlugin<RollsRoyceMT30>
      RollsRoyceMT30::plugin(RollsRoyceMT30::className_);

RollsRoyceMT30::RollsRoyceMT30()
{
      setDefaults();
}

RollsRoyceMT30::RollsRoyceMT30(const string & name)
{
      setName(name);
      setDefaults();
}

void RollsRoyceMT30::setDefaults()
{
      //Add models to the ResistiveNetworkGroupModel arrays
      addModel(fuelValve_);

      addModel(IPC_);
      addModel(compressorNode_);
      addModel(HPC_);
      addModel(combustor_);
      addModel(HPT_);
      addModel(upperTurbineNode_);
      addModel(IPT_);
      addModel(lowerTurbineNode_);
      addModel(FPT_);

      addModel(lowPressureShaft_);
      addModel(highPressureShaft_);

      //Settings for Intermediate Pressure Compressor (IPC_)
      IPC_.setDesignPressureRatio(24/4.2);
      IPC_.setDesignSpeed(7500);
```

```
//Settings for High Pressure Compressor (HPC_)
HPC_.setDesignPressureRatio(4.2);
HPC_.setDesignSpeed(10000);

//Settings for Combustor
combustor_.setDesignEfficiency(1.0);
combustor_.setDesignFuelFlowIn(2.07);
combustor_.setEfficiency(1.0);

//Settings for Low Pressure Shaft
lowPressureShaft_.setInertia(550);
lowPressureShaft_.setTurningGearSpeed(5);
lowPressureShaft_.setSpeed(7500);

//Settings for High Pressure Shaft
highPressureShaft_.setInertia(620);
highPressureShaft_.setTurningGearSpeed(5);
highPressureShaft_.setSpeed(10000);

//Settings for Fuel Valve
double fuelDesignFlow = 2.07;
double fuelDesignValvePosition = 0.1;
double fuelDesignPressureDifference = (4000000-2431800);

fuelValve_.setDesignOpenFlowRate(fuelDesignFlow /
      fuelDesignValvePosition);
fuelValve_.setDesignPressureDifference(
      fuelDesignPressureDifference);
fuelValve_.setValvePosition(10);

//Model connections
IPC_.setOutletEffortModel(compressorNode_);
compressorNode_.addOutletFlowModel(HPC_);
combustor_.setAirInletFlowModel(HPC_);
combustor_.setFuelInletFlowModel(fuelValve_);
combustor_.setAirOutletFlowModel(HPT_);
HPT_.setOutletEffortModel(upperTurbineNode_);
upperTurbineNode_.addOutletFlowModel(IPT_);
IPT_.setOutletEffortModel(lowerTurbineNode_);
lowerTurbineNode_.addOutletFlowModel(FPT_);

//Shaft connections
lowPressureShaft_.addModel(IPC_);
lowPressureShaft_.addModel(IPT_);
highPressureShaft_.addModel(HPC_);
highPressureShaft_.addModel(HPT_);

//Fluid property
isFluidPerfect_ = false;


//Set default write variables
```

```
	for(size_t x = 0; x < allModels_.size(); ++x)
		addWriteVariableList(
			allModels_[x]->getWriteVariableList());
}

void RollsRoyceMT30::initialize()
{
#ifdef INCLUDE_DEBUG_STATEMENTS
	DEBUG.entering(localDebugLevel_, className_, name_,
		"initialize", "");
	DEBUG.input("isFluidPerfect_", isFluidPerfect_);
#endif

	//Inlet settings for design point
	inletAirFluid_->updatePropsPT(101325, 288.15);

	//Apply fluids to the models
	IPC_.setFluid(inletAirFluid_);
	HPC_.setFluid(inletAirFluid_);
	combustor_.setFluid(outletAirFluid_);
	HPT_.setFluid(outletAirFluid_);
	IPT_.setFluid(outletAirFluid_);
	FPT_.setFluid(outletAirFluid_);

	compressorNode_.setFluid(inletAirFluid_);
	upperTurbineNode_.setFluid(outletAirFluid_);
	lowerTurbineNode_.setFluid(outletAirFluid_);

	fuelValve_.setFluid(inletFuelFluid_);

	//Set the design efficiencies
	if(isFluidPerfect_)
	{
		IPC_.setDesignEfficiency(0.8859702677331826);
		HPC_.setDesignEfficiency(0.8904429503214635);
		HPT_.setDesignEfficiency(0.9089229485360016);
		IPT_.setDesignEfficiency(0.9079108334492134);
		FPT_.setDesignEfficiency(0.9158218656215227);
	}
	else
	{
		IPC_.setDesignEfficiency(0.8846460640329019);
		HPC_.setDesignEfficiency(0.8880828547244052);
		HPT_.setDesignEfficiency(0.9237636357416308);
		IPT_.setDesignEfficiency(0.9230073169094473);
		FPT_.setDesignEfficiency(0.9309009879731457);
	}

	//Calculate the design parameters of the models
	IPC_.setDesignFlow(110.93);
	IPC_.setDesignPressureIn(101325);
	IPC_.setDesignEnthalpyIn(inletAirFluid_->getEnthalpy());
	IPC_.calculateDesignParameters();
```

```
      HPC_.setDesignPressureIn(IPC_.getDesignPressureOut());
      HPC_.setDesignEnthalpyIn(IPC_.getDesignEnthalpyOut());
      HPC_.setDesignFlow(IPC_.getDesignFlow());
      HPC_.calculateDesignParameters();

      combustor_.setDesignPressure(HPC_.getDesignPressureOut());
      combustor_.setDesignEnthalpyIn(HPC_.getDesignEnthalpyOut());
      combustor_.setDesignAirFlowIn(HPC_.getDesignFlow());
      combustor_.calculateDesignParameters();

      HPT_.setDesignPressureIn(combustor_.getDesignPressure());
      HPT_.setDesignEnthalpyIn(combustor_.getDesignEnthalpyOut());
      HPT_.setDesignFlow(combustor_.getDesignAirFlowOut());
      HPT_.setDesignPower(-(HPC_.getDesignPower()));
      HPT_.setDesignSpeed(HPC_.getDesignSpeed());
      HPT_.calculateDesignParameters();

      IPT_.setDesignPressureIn(HPT_.getDesignPressureOut());
      IPT_.setDesignEnthalpyIn(HPT_.getDesignEnthalpyOut());
      IPT_.setDesignFlow(HPT_.getDesignFlow());
      IPT_.setDesignPower(-(IPC_.getDesignPower()));
      IPT_.setDesignSpeed(IPC_.getDesignSpeed());
      IPT_.calculateDesignParameters();

      FPT_.setDesignPressureIn(IPT_.getDesignPressureOut());
      FPT_.setDesignEnthalpyIn(IPT_.getDesignEnthalpyOut());
      FPT_.setDesignFlow(IPT_.getDesignFlow());
      FPT_.setDesignPower(36000);
      FPT_.setDesignSpeed(3600);
      FPT_.calculateDesignParameters();

      //Call the base class initialization
      ResistiveNetworkGroupModel::initialize();


#ifdef INCLUDE_DEBUG_STATEMENTS
      DEBUG.exiting();
#endif
}

void RollsRoyceMT30::setInletAirEffortModel(ThermalFluidEffortModel *
      inletAirEffortModel)
{
      IPC_.setInletEffortModel(inletAirEffortModel);
}

void RollsRoyceMT30::setInletAirEffortModel(ThermalFluidEffortModel &
      inletAirEffortModel)
{
      setInletAirEffortModel(&inletAirEffortModel);
}
```

```
void RollsRoyceMT30::setInletFuelEffortModel(ThermalFluidEffortModel *
      inletFuelEffortModel)
{
      fuelValve_.setInletEffortModel(inletFuelEffortModel);
}

void RollsRoyceMT30::setInletFuelEffortModel(ThermalFluidEffortModel &
      inletFuelEffortModel)
{
      setInletFuelEffortModel(&inletFuelEffortModel);
}

void RollsRoyceMT30::setOutletAirEffortModel(ThermalFluidEffortModel *
      outletAirEffortModel)
{
      FPT_.setOutletEffortModel(outletAirEffortModel);
}

void RollsRoyceMT30::setOutletAirEffortModel(ThermalFluidEffortModel &
      outletAirEffortModel)
{
      setOutletAirEffortModel(&outletAirEffortModel);
}

void RollsRoyceMT30::setShaft(Shaft * shaft)
{
      FPT_.setShaft(shaft);
}

void RollsRoyceMT30::setShaft(Shaft & shaft)
{
      setShaft(&shaft);
}

void RollsRoyceMT30::setName(const string & name)
{
      name_ = name;

      IPC_.setName(name + "_IPC");
      HPC_.setName(name + "_HPC");
      combustor_.setName(name + "_Comb");
      HPT_.setName(name + "_HPT");
      IPT_.setName(name + "_IPT");
      FPT_.setName(name + "_FPT");

      compressorNode_.setName(name + "_N1");
      upperTurbineNode_.setName(name + "_N2");
      lowerTurbineNode_.setName(name + "_N3");

      lowPressureShaft_.setName(name + "_LowPressureShaft");
      highPressureShaft_.setName(name + "_HighPressureShaft");

      fuelValve_.setName(name + "_FuelValve");
```

```
}

void RollsRoyceMT30::setWriteFlag(bool writeFlag)
{
      writeFlag_ = writeFlag;

      bool modelWriteFlag;
      modelWriteFlag = writeFlag;

      IPC_.setWriteFlag(modelWriteFlag);
      HPC_.setWriteFlag(modelWriteFlag);
      combustor_.setWriteFlag(modelWriteFlag);
      HPT_.setWriteFlag(modelWriteFlag);
      IPT_.setWriteFlag(modelWriteFlag);
      FPT_.setWriteFlag(modelWriteFlag);

      compressorNode_.setWriteFlag(modelWriteFlag);
      upperTurbineNode_.setWriteFlag(modelWriteFlag);
      lowerTurbineNode_.setWriteFlag(modelWriteFlag);

      lowPressureShaft_.setWriteFlag(modelWriteFlag);
      highPressureShaft_.setWriteFlag(modelWriteFlag);

      fuelValve_.setWriteFlag(modelWriteFlag);
}

void RollsRoyceMT30::setInletAirFluid(DTMSFluid * fluid)
{
      inletAirFluid_ = fluid;
}

void RollsRoyceMT30::setInletAirFluid(DTMSFluid & fluid)
{
      setInletAirFluid(&fluid);
}

void RollsRoyceMT30::setInletFuelFluid(DTMSFluid * fluid)
{
      inletFuelFluid_ = fluid;
}

void RollsRoyceMT30::setInletFuelFluid(DTMSFluid & fluid)
{
      setInletFuelFluid(&fluid);
}

void RollsRoyceMT30::setOutletAirFluid(DTMSFluid * fluid)
{
      outletAirFluid_ = fluid;
}

void RollsRoyceMT30::setOutletAirFluid(DTMSFluid & fluid)
{
```

```
	setOutletAirFluid(&fluid);
}

void RollsRoyceMT30::setIsFluidPerfect(bool isFluidPerfect)
{
	isFluidPerfect_ = isFluidPerfect;
}

void RollsRoyceMT30::setSpeed(double speed)
{
	FPT_.setSpeed(speed);
}

double RollsRoyceMT30::getSpeed()
{
	return FPT_.getSpeed();
}

void RollsRoyceMT30::setPower(double power)
{
	FPT_.setPower(power);
}

double RollsRoyceMT30::getPower()
{
	return FPT_.getPower();
}

void RollsRoyceMT30::setInertia(double inertia)
{
	FPT_.setInertia(inertia);
}

double RollsRoyceMT30::getInertia()
{
	return FPT_.getInertia();
}

void RollsRoyceMT30::set(const DTMSData & variable, double value)
{
	if(variable == VALVE_POSITION)
		fuelValve_.setValvePosition(value);

	else if(variable == ENTHALPY_IN)
		IPC_.set(ENTHALPY_IN, value);

	else
		FPT_.set(variable, value);
}

double RollsRoyceMT30::get(const DTMSData & variable)
{
	if(variable == VALVE_POSITION)
```

```
            return fuelValve_.getValvePosition();

      else if(variable == ENTHALPY_IN)
            return IPC_.get(ENTHALPY_IN);

      else
            return FPT_.get(variable);
}

}
```

References

[1] Alexandrescu, Andrei. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[2] Ashmore, C., "MT30 flat-rated at 36,000 kW and 40% efficiency up to 26ºC ambient." GAS TURBINE WORLD, v. 34, n. 1, pp. 32-35. February-March 2004.

[3] Carroll, Brian Christopher, "Improved Thermal Management of an All-Electric Ship through Modeling and Simulation." Master's Thesis. The University of Texas at Austin, 2004.

[4] Department of Defense, "Militarily Critical Technologies List; Section 7: Energy Systems Technology." Report. Office of the Under Secretary of Defense—Acquisition, Technology, and Logistics, Pentagon, VA, September 2005.

[5] "ESRDC Background." ESRDC, 2007. <http://esrdc.caps.fsu.edu/esrdc.background.html>. 1 December 2009.

[6] "ESRDC Consortium Thrusts." ESRDC, 2007. <http://esrdc.caps.fsu.edu/thrusts.html>. 1 December 2009.

[7] "Fact Sheet: MT30 marine gas turbine." Rolls-Royce, 2003. <http://www.rolls-royce.com/marine/product/gasturbines/mt30/downloads/mt30factsheet.pdf>. 25 June 2008.

[8] Floyd, J., Hunt, S., Williams, F., and Tatem, P. "A Network Fire Model for the Simulation of Fire Growth and Smoke Spread in Multiple Compartments with Complex Ventilation." Journal of Fire Protection Engineering, Vol. 15, No. 3, pp. 199-229. August 2005.

[9] Floyd, J., Hunt, S., Williams, F., and Tatem, P. "Fire and Smoke Simulator (FSSIM) Version 1 - Theory Manual." Naval Research Laboratory. 31 March 2009.

[10] Haag, Scott Thomas, "Steady-State and Dynamic Simulation of Large Thermal Systems." Master's Thesis. The University of Texas at Austin, December 2005.

[11] Harman, R., *Gas Turbine Engineering*, John Wiley & Sons, Inc., New York, NY, 1981.

[12] Haupt, T., Williams, F., Henley, G., Tatem, P., Parihar, B., Floyd, J., Kirkland, R., and Scheffey, J. "Graphical User Interface Version 2.10 with Fire and Smoke Simulation Model (FSSIM) Version 1.2." Naval Research Laboratory. October 2009.

[13] Hewlett, Patrick Thomas, "Implementation of an In-House Framework for Dynamic Assessment of Thermal Load Management Strategies aboard Navy Surface Ships." Master's Thesis. The University of Texas at Austin, December 2008.

[14] Holsonback, Christopher Ryan, "Dynamic Thermal-Mechanical-Electrical Modeling of the Integrated Power System of a Notional All-Electric Naval Surface Ship." Master's Thesis. The University of Texas at Austin, May 2007.

[15] "How to design a simple C++ object factory?" Stack Overflow. <http://stackoverflow.com/questions/333400/how-to-design-a-simple-c-object-factory>. 3 April 2009.

[16] Johnson, A., *Biological Process Engineering: An Analogical Approach to Fluid Flow, Heat Transfer, and Mass Transfer Applied to Biological Systems*. Wiley-IEEE, New York, 1998.

[17] Meynard, Yves, "When enum Just Isn't Enough: Enumeration Classes for C++". Dr. Dobb's Portal: The World of Software Development. <http://www.ddj.com/cpp/184403955>. 3 April 2009.

[18] National Institutes of Standards and Technology, "NIST Reference Fluid Thermodynamic and Transport Properties—REFPROP." Computer Program, 2007.

[19] "Newton-Cotes formulas." Wikipedia. Wikimedia Foundation, Inc, 2009. <http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas>. 6 January 2009.

[20] Paullus, Patrick Earl, "Creation of a Modeling and Simulation Environment for Thermal Management of an All-Electric Ship." Master's Thesis. The University of Texas at Austin, December 2007.

[21] "PID Controller." Wikipedia. Wikimedia Foundation, Inc, 2009. <http://en.wikipedia.org/wiki/PID_controller>. 17 November 2009.

[22] Pierce, Ryan J., "Thermal Management of the All-Electric Ship." Master's Thesis. Naval Postgraduate School, 2004.

[23] Pruske, Matthew Andrew, "Thermal-Electrical Co-Simulation of Shipboard Integrated Power Systems on an All-Electric Ship." Master's Thesis. The University of Texas at Austin, August 2009.

[24] Rowen, W., "Simplified Mathematical Representations of Heavy-Duty Gas Turbines." Journal of Engineering for Power, vol. 105, pp. 865-869. October 1983.

[25] Schulz, N., Hebner, R., Dale, S., Dougal, R., Sudhoff, S., Zivi, E., and Chryssostomidis, C. "The U.S. ESRDC Advances Power System Research for Shipboard Systems." Universities Power Engineering Conference, 2008. UPEC 2008. 43rd International. September 2008.

[26] Sutter, Herb and Alexandrescu, Andrei. *C++ coding standards: 101 rules, guidelines, and best practices*. Addison-Wesley, Boston, MA, 2005.

[27] Sutter, Herb. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley, Reading, MA, 2000.

[28] Sutter, Herb. *Exceptional C++ style: 40 new engineering puzzles, programming problems, and solutions*. Addison-Wesley, Boston, MA, 2005.

[29] Sutter, Herb. *More exceptional C++: 40 new engineering puzzles, programming problems, and solutions*. Addison-Wesley, Boston, MA, 2002.

[30] Webb, Tyler Wybest, "Thermal Management of Pulsed Loads on an All-Electric Ship." Master's Thesis. The University of Texas at Austin, August 2006.

[31] Weisstein, Eric W. "Backward Difference." MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/BackwardDifference.html>. 6 January 2009.

[32] Weisstein, Eric W. "Newton-Cotes Formulas." MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/Newton-CotesFormulas.html>. 6 January 2009.

[33] Zerby, M., "Thermal Management for the Electric Warship." 2006 Electric Machine Technology Symposium, 2006.

Vita

Michael Stephen Pierce was born June 17, 1985, in Waco, Texas, to Ben Pierce and Marlene Tyrrell. Raised in Hewitt, a suburb of Waco, he attended the elementary, intermediate, middle, and high schools of the Midway Independent School District. Michael graduated from Midway High School in 2003 with salutatorian honors and ventured on to attended Southern Methodist University in Dallas, Texas as a member of the Dean's Scholars Program and the Industry Scholars Intern Program. Focusing on a double-major in Mechanical Engineering and Mathematics with a minor in Computer Science, Michael completed his degrees in the spring of 2007 as the standard bearer for the Bobby B. Lyle School of Engineering with magna cum laude honors. In the summer of 2007, Michael contributed to the graduate research programs of both Dr. Paul S. Krueger of Southern Methodist University and Dr. Thomas Kiehne of the University of Texas at Austin. For the fall semester, Michael officially enrolled in the Mechanical Engineering graduate program of the Cockrell School of Engineering at the University of Texas at Austin and immediately joined the naval thermal management research program headed by Dr. Thomas Kiehne, completing his graduate work in December of 2009.

Permanent address: Michael S. Pierce
103 Canyon Rd.
Georgetown, TX 78628

Mobile: (214) 738-4359
michaelspierce@gmail.com

This thesis was typed by the author.